OPTIMIZATION OF CONDITION TESTING FOR MULTI-JOIN TRIGGERS IN ACTIVE DATABASES

By

SREENATH BODAGALA

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1998

To
My Parents

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor Dr. Eric N. Hanson for giving me an opportunity to work on this challenging topic and for providing continuous guidance, advice, encouragement and support throughout the course of this research work. He has been a constant source of inspiration throughout.

I would like to thank Dr. Stanley Su, Dr. Sharma Chakravarthy, Dr. Herman Lam, and Dr. Haniph Latchman for serving on my committee and for their valuable feedback.

I would like to thank all my friends for their help and for making my stay at Gainesville a memorable one. Thanks are also due to our dear secretary Ms. Sharon Grant, for maintaining a well administered Database Systems Research and Development Center.

Finally, I would like to thank my parents, my brother and my sister, without whose love, support and constant encouragement this work would not have been possible.

This work was funded by the National Science Foundation under grant number IRI-9318607.

TABLE OF CONTENTS

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

OPTIMIZATION OF CONDITION TESTING FOR MULTI-JOIN TRIGGERS IN ACTIVE DATABASES

By

Sreenath Bodagala

August, 1998

Chairman: Dr. Eric N. Hanson
Major Department: Computer and Information Science and Engineering

An active database management system is a database management system extended with trigger processing capabilities. For an active database system to be successful, it must be able to provide the trigger processing facility without significant performance overhead. This dissertation investigates various strategies to improve the performance of the trigger condition testing component of an active database system. Most of the techniques proposed in the litertaure for testing trigger conditions in a database environment are based on the idea of using an existing query processing system. Even though these techniques work well for simple triggers, they incur a lot of redundant computation for multi-join triggers.

Discrimination networks perform incremental testing of trigger conditions and have the potential to outperform other techniques for multi-join triggers. This dissertation explores the application of a generalized version of TREAT and Rete discrimination networks (called *Gator* networks) for trigger condition testing in active databases. Gator network optimizer has been developed to choose an efficient Gator network for testing the condition of a trigger, given information about the structure of the trigger, database size, attribute cardinality and update frequency distribution. The optimizer uses randomized search techniques to deal with the problem of a large search space.

Experimental results show that Gator networks perform significantly better than TREAT and reasonably better than Rete in a plentiful-buffer environment. The experiments also reveal that the optimized Gator network shape in most of the experiments is neither pure Rete nor TREAT, but an intermediate form having a few beta nodes. Validation of Gator network cost functions was performed, showing a strong correlation between the expected and the actual cost of a Gator network.

This dissertation also investigates the Multiple Rule Optimization (MRO) problem. A new architecture to perform MRO has been proposed. A search strategy based on randomized algorithms along with a set of heuristics to reduce the search space, has been presented to find an efficient global strategy for a set of rules. Experimental results show that the search strategy based on randomized algorithms generates reasonably good solutions.

CHAPTER 1
INTRODUCTION

A database management system extended with trigger processing capabilities is called an active database management system. Conventional database systems are passive: they perform actions only when explicitly requested by a user. Active database systems, in addition to performing the activities of a database management system, have the ability execute actions automatically when specified events occur and/or when the database state satisfies a specified condition. This *active* behavior is made possible with the help of *rules* or *triggers*. Rules have various applications in a database environment: they can be used to ensure integrity constraints, provide view maintenance, support condition monitoring applications, statistics gathering, version management and authorization. Implementing these features in a conventional database system involves either encoding the logic in application programs or writing special purpose subsystems to perform the task. Rules can also be used to support advanced applications like data-intensive expert systems and workflow management, which are beyond the scope of passive database systems.

Because of their wide applications, active database systems have received considerable attention in the database research community in the past 10 years. Many prototype active database systems have been built [50, 14, 61, 5, 8, 12] and also, many commercial database systems [20, 38, 19, 62] support a rule processing facility.

The E-C-A model [8] is the most widely used model to specify rules in a database environment. The format of an ECA rule is given below:

```
on event
if condition
then action
```

An ECA rule is triggered when the event occurs in a database. The condition of the trigger is evaluated and if the condition is true, the specified action is executed. To support ECA rules, a database system must be able to detect events, evaluate rule conditions and take actions when the conditions are satisfied.

One of the desirable features of an active database system is that it should be able to process rules without significant performance overhead. The process of rule condition testing, which involves matching a rule condition against a disk-resident database state, is the most time-consuming part of rule processing. Since a rule condition needs to be tested for every database event, efficient rule condition testing is crucial to the performance of active databases.

This dissertation investigates various techniques to provide efficient support for complex or multi-join triggers in an active database system. Section 1.1 presents various rule condition testing strategies that have been explored in the literature. Most of these techniques are based on the idea of using an existing query optimizer to generate a plan for evaluating a rule condition. The generated plan is executed on the database for all relevant database events. This technique is simple and works well for simple rule conditions. However, since it involves iterating over the entire data set for all database events, it is not efficient for complex or multi-join rule conditions.

Discrimination networks have successfully been used in doing pattern matching of production rules. Discrimination networks perform incremental testing of rules and have the potential to test rule conditions efficiently in a database environment also. Rete and TREAT are the commonly used discrimination networks in production systems. Performance studies have shown that, in a database environment, neither of these two provide an optimal network structure under all circumstances [60]. Gator discrimination networks [13] have the potential to perform well in all cases.

However, the number of possible Gator networks that can do pattern matching for a rule condition is very large. This necessitates the development of a Gator network optimizer that can find an efficient Gator network for a rule. The search space of a Gator network optimizer is much higher than that of a query optimizer. An experimental study showed that it was not feasible to use a dynamic programming approach for optimizing Gator networks when the rule condition contained more than seven relations [18]. This dissertation explores a randomized algorithms-based strategy to optimize Gator networks. It also reports an experimental study that was conducted to compare the actual rule condition testing performance of Gator, Rete and TREAT networks.

This dissertation also investigates the problem of performing efficient pattern matching for a collection of triggers in an active database system. This problem is named Multiple Rule Optimization (MRO). This is the problem of finding discrimination networks for a set of triggers such that the total cost of pattern matching for all the triggers is minimal. Rules that are defined in the context of a single or related applications tend to have common computation and MRO takes advantage of that to improve the trigger condition testing performance.

MRO is similar to the problem of Multiple Query Optimization. The tasks to be performed to achieve MRO include identifying the common expressions between different rule conditions and performing search among the discrimination networks of rules. This dissertation proposes a new strategy to solve the MRO problem. Next, it proposes a search strategy based on randomized algorithms and a set of heuristics to reduce the search space of the search algorithm. The results of an experimental study have been reported to show the effectiveness of the proposed heuristics and the search strategy.

1.1 Trigger Condition Testing Strategies

The various techniques proposed for trigger condition testing in the literature are the following: 1. Brute force approach, 2. Marking scheme, 3. Compilation approach, and 4. Discrimination networks. These techniques are explained in detail next.

Brute force approach: In this approach, each relation in the database is associated with a list of rules that affect that relation. For every update to a relation, the conditions of all the rules in the list are tested sequentially. The plan for evaluating the condition of a rule is generated by using an existing query optimizer. This scheme is simple and works well for small number of rules. However, as the number of rules and/or the complexity of rules increases, it becomes less desirable. The reason is that this approach involves iterating over the entire data set for every database event and it is possible to improve upon that in a lot of cases. HiPAC [8], Ode [12] and Starburst [61] are some of the active database systems that use this technique.

Marking scheme: In this scheme, the condition of a rule is evaluated against the database and a marker is placed on each of the tuples satisfying its rule condition. Thus, each tuple is associated with zero or more markers identifying the rules whose conditions are satisfied by that tuple. When a tuple is touched during query processing, rules corresponding to the markers placed on it are activated. POSTGRES [52, 50, 53, 47, 48, 49] uses this technique. Consider, for example, the following POSTGRES rule.

```
define rule ExampleRule1
on retrieve to Emp.salary
where Emp.Name="Bob"
then do append to Audit (Name=current.Name, user=user())
```

ExampleRule1 appends an audit record to the Audit table whenever Bob's salary is accessed. POSTGRES implements the above rule by placing a marker, corresponding to the above rule, on the tuples of the Emp table satisfying the qualification "Emp.Name=Bob".

This scheme is very efficient for single-table rules since the run-time activity for detecting the rules to be triggered is minimal. However, this scheme requires more space especially when there are lot of tuples in a database that can satisfy a rule condition. Also, markers need to be maintained in presence of updates to the database and that causes a lot of overhead [52].

Discrimination Networks: In this approach, rule condition testing is performed by using discrimination networks. Discrimination networks have long been used in doing pattern matching of production rules. Discrimination networks utilize the temporal redundancy property of data to speed up rule condition testing. Discrimination networks store state between database transitions and utilize that information to detect new matches to rules. Rete [11] and TREAT [34] are the commonly used discrimination networks in production systems. Gator (Generalized TREAT/Rete) Networks are proposed in [13]. More details about discrimination networks are given in section 1.2. Ariel [14] uses discrimination networks for doing pattern matching of rules.

Compilation Approach or Query Rewrite Approach: In this approach, database operations are modified to take into account the effect of rules. Hence, this scheme requires no run-time activity to detect events or to evaluate rule conditions. But, this scheme is applicable only in cases where the database events are detectable at compilation time and the database language is amenable for modification to include the effects of rules. This technique has been implemented in KBMS [54, 55, 56] and POSTGRES [52].

The following two examples illustrate the query rewriting approach used in POSTGRES. Consider the following POSTGRES rule (from [62]):

```
define rule ExampleRule2
```

```
    on retrieve to Emp.manager
    instead retrieve Dept.manager
    where current.dept=Dept.name
```

and the query

```
    retrieve(Emp.name, Emp.manager)
```

POSTGRES rewrites the above query to:

```
    retrieve (EMP.name, Dept.manager)
    where Emp.Dept=Dept.dname
```

This query is then optimized by the query optimizer of POSTGRES and executed. If the marking scheme is used, it will cause the rule condition to be evaluated once for every tuple in the Emp relation. Hence, the query rewriting technique is superior to the marking scheme for this example.

The following example illustrates a case where the marking scheme may be better than the query rewriting technique.

```
    define rule ExampleRule3
    on retrieve to Emp.Salary
    where current.name = "Bob"
    do instead retrieve (emp.salary)
    where emp.name = "Dan"
```

user query:

```
    retrieve (emp.salary)
    where R1.age > 30
```

The POSTGRES Query Rewrite System replaces the above query with the following two queries:

```
    retrieve (emp.salary)
    where emp.age > 30 and
    not (emp.name="Bob")


    retrieve (e.salary)
    from e in emp
    where emp.age > 30
    and emp.name = "Bob"
    and e.name = "Dan"
```

The POSTGRES optimizer optimizes both the queries generated above and executes them in place of the user query. Since only a few tuples are expected to satisfy the rule condition (tuples of Bob with age > 30), the marking scheme may be more efficient than the rewriting approach for this example (here, the rewriting scheme causes two queries to be executed).

POSTGRES decides the choice of implementation (marking scheme or query rewriting scheme) for a rule based on the number of tuples that are expected to satisfy the rule condition. If the expected number of tuples is less than a cutoff value then the marking scheme is used; otherwise the rewriting scheme is used. Refer to [35] for details on choosing the cutoff value for a rule.

The following example illustrates the compilation approach used in KBMS [56].

```
Class Dummy {
attributes:
method specifications:
method M1()
Rules:
ExampleRule4
}


Rule ExampleRule4;
Before method M1()
Condition X
Action
```

Method M1() is translated to:

```
M1() {
   If X then Action
   Code to implement the original M1()
}
```

The rule specification and the class definition are given in NCL (NIIP Common Language) [58]. Execution of class methods form triggering events in KBMS. ExampleRule4 specifies that before the method M1 of class Dummy is evaluated, the condition X should be checked. If the condition is satisfied then the specified action should be performed. KBMS implements it by modifying the code of method M1 as shown above. The new method will have the code to evaluate the rule condition and execute the specified action (if necessary) before processing the original method.

1.2 Types of Discrimination Networks

Discrimination networks are tree structures with materialized nodes, constructed to test the conditions of rules efficiently. Based on the shape of the tree structure, discrimination networks can be classified into Rete, TREAT and Gator networks. The Rete algorithm was proposed by Forgy [11] for doing pattern matching of rules in OPS5 [3]. TREAT algorithm, proposed by Miranker [34], is a modification of Rete for better performance. Gator networks, proposed by Hanson [13], have the potential to outperform both Rete and TREAT networks.

In this work, it is assumed that the condition of a rule has the same structure as a relational database query having selection and join conditions on one or more tables. The condition of a rule can be represented by a condition graph which has a node for each tuple variable and an edge for each join condition specified in the rule condition. The condition graph drawn for a rule condition is called a *Rule Condition Graph (RCG).*

One possible set of Rete, TREAT and Gator networks for an arbitrary rule RuleDummy below are shown in Figures 1.1, 1.2 and 1.3 respectively. Details of rule condition testing using Rete, TREAT and Gator networks are given in the next subsection.

```
define rule RuleDummy
if R1.b = const1
and R2.a = const2
and R1.a = R2.b and
and R2.c = R3.d and
and R3.c = R4.e and
and R3.b = R5.a
then Action
```

1.2.1 Rete Networks

Rete networks are binary tree structures. Rete networks consist of the following types of nodes:

t-const nodes These nodes test selection conditions. A t-const node is created for each relation (or for each tuple variable, to be more precise) in the rule condition.

alpha nodes These nodes store the results of t-const nodes. One alpha node is created for each t-const node.

and nodes These nodes join the tokens of either two alpha nodes or a beta node and an alpha node.

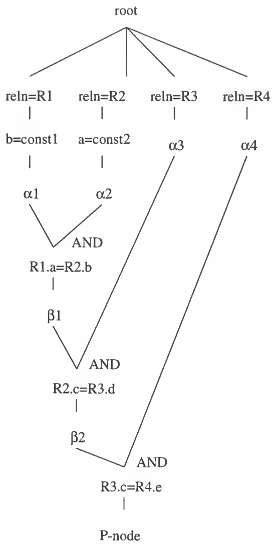


Figure 1.1. Rete network for RuleDummy

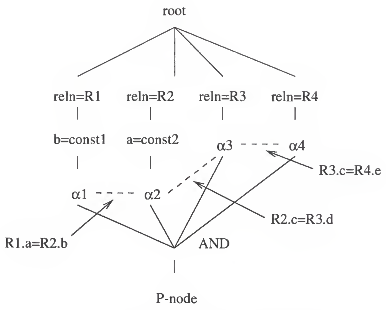


Figure 1.2. TREAT network for RuleDummy

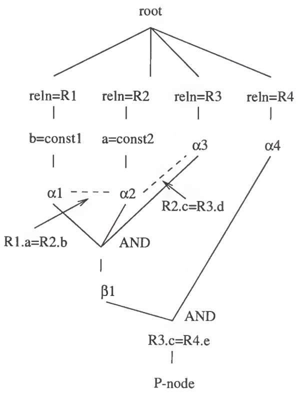


Figure 1.3. Gator network for RuleDummy

beta nodes These nodes store the results of and nodes. One beta node is created for each and node.

P-node This node holds the tuples that match an entire rule condition.

root node one root node per rule.

Tokens representing the changes to the working memory enter a Rete network through its root node. The root node passes the incoming tokens to an appropriate t-const node. Each t-const node is associated with a selection condition. At a t-const node, tokens are tested against the selection condition that is associated with it. If a token satisfies the selection condition, it is inserted into the alpha node corresponding to that t-const node and a copy of it is then passed to the parent of the alpha node. At an AND node, a token arriving from one of its child nodes is joined with the tokens of its other child node. Matching tokens are inserted into the beta node corresponding to that AND node. Copies of these tokens are then propagated to the parent of the beta node. Tokens reaching the P-node activate the rule.

The advantages of Rete networks are the following: 1. Rete performs matching without iterating over the entire working memory. It stores large amount of information between transitions and uses that information to do matching. 2. Similar Rete sub-networks can be shared among different rules.

The disadvantages of Rete networks are the following: 1. Deletions are expensive. When a tuple is deleted, all the tokens corresponding to that tuple are to be deleted from the entire network. 2. The size of beta nodes can be huge. Hence, update operations on a beta node are often going to be expensive.

1.2.2 TREAT Networks

The TREAT network was proposed by Miranker [34]. A TREAT network contains no beta nodes. It contains a root node, t-const nodes, alpha nodes, a P-node and an AND node. There are as many t-const nodes as the number of relations in a rule condition. An alpha node is associated with each t-const node. There is a single AND node and it joins the tokens of all the alpha nodes in the network. A P-node holds the tokens matching the entire rule condition.

The motivation for the TREAT network was the observation by McDermott [32] that, the retesting cost will be less than the cost of maintaining the state required for condition relationship. Token propagation in TREAT network is similar to that in Rete. Any token arriving at an AND node is joined with all the other alpha nodes and the resulting tokens are placed in the P-node. The join order plan of a node gives the order in which a token arriving at that node is to be joined with the other alpha nodes. The join order plan for the alpha nodes can be generated either by using heuristics or by using query optimization algorithms. The join order plan can be constructed either during compilation time or it can be constructed on-the-fly during pattern matching.

A-TREAT [14] is an adaptation of the TREAT algorithm for the database environment. In A-TREAT, there are two types of alpha nodes: stored and virtual. A stored alpha node is the same as that of an alpha node in the TREAT algorithm. A virtual alpha node differs from the stored alpha node in the following way: it does not hold the tokens that match the selection condition in its associated t-const node. The tokens are computed on demand from the base relation by applying the selection predicate. The reason for introducing the virtual alpha nodes is the following: a stored alpha node holds the tuples of the base relation that match the selection predicate. If the selectivity of the predicate is poor, a stored alpha node might have to hold a large fraction of the tuples of the base relation. This is not only redundant but also not acceptable in a database environment since the relation size can be huge. Virtual alpha nodes are introduced to replace stored alpha nodes in those situations. Also, virtual alpha nodes can take advantage of the indexes on the base relation to process tokens. An alpha node may be made virtual whenever storing the tokens in it is not beneficial. Token propagation in presence of virtual alpha nodes is described in [14]. A-TREAT also uses a selection predicate index to speed up selection predicate testing. [16] gives more details on selection predicate indexing.

1.2.3 Gator Networks

Performance studies in [60] indicated that, in a database environment, TREAT usually outperforms Rete but Rete is better than TREAT in a few cases where the frequency distribution of updates to different relations in the rule condition is skewed. Gator networks [13], if properly tuned, have the potential to perform well in all cases.

Gator (Generalized TREAT/Rete) networks are generalized tree structures. Gator networks contain the same type of nodes as that of Rete networks but with the following difference: an AND node in a Gator network can join any number of alpha or beta nodes. Also, an alpha node in a Gator network can either be a stored alpha node or a virtual alpha node.

There are many Gator networks that can do pattern matching for a given rule. Rete and TREAT networks are special cases of Gator networks.

Token propagation in Gator networks is similar to that of Rete and TREAT. A token coming into an AND node is joined with all the other child nodes (alpha or beta) of that AND node. The order in which the nodes are joined is controlled by the join order plan stored in each node. Matched tokens are inserted into the beta nodes.

From now on, for the sake of simplicity, we assume that the selection predicates of t-const nodes are stored directly in the alpha nodes. This assumption does not compromise the structure of the discrimination networks in any way. Similarly, we also assume that the join predicates of AND nodes are stored directly in the beta nodes.

1.3 Conclusion

This chapter presented various techniques that were proposed for rule condition testing in the literature. Rule condition testing using discrimination networks was discussed in detail. Discrimination networks perform incremental testing of rules and have the potential to perform well for multi-join triggers. The following chapters explore the discrimination network-based approach for the condition testing of multi-join triggers. Chapters 2, 3 and 4 concentrate on the problem of finding an optimal Gator discrimination network for a rule. Chapter 5 compares the performance of

Gator with that of Rete and TREAT networks and also, it validates the Gator network cost model.

The rest of this dissertation is organized as follows.

Chapter 2 discusses the problem of deciding the status of alpha nodes in a Gator network. Possible choices for the status of an alpha node are presented. A new strategy is proposed, based on cost functions, that can decide the status of alpha nodes in a Gator network with minimum overhead during the Gator network optimization process itself.

Chapter 3 discusses the problem of estimating the cost of a Gator network of a rule, given information about the structure of the rule, database size, attribute cardinalities and update frequency distribution. Two cost models are proposed based on the amount of buffer space that is available for trigger processing. The details of these cost models are given in this chapter.

Chapter 4 investigates the Gator network optimization problem. An upper bound on the search space of the problem is derived. A strategy for optimizing Gator networks, based on randomized algorithms, is presented. Three randomized algorithms, II, SA and TPO are applied for the Gator network optimization problem. The details of these algorithms and the results of a performance evaluation of them are presented. An explanation of these results based on the possible search space of the problem is given.

Chapter 5 presents the details of various experiments conducted to study the relative performance of Gator, Rete and TREAT networks. Details about the accuracy of the proposed Gator network model are presented and improvements are suggested.

Chapter 6 investigates the problem of Multiple Rule Optimization. A new approach to solve the Multiple Rule Optimization problem is presented. A search strategy based on randomized algorithms along with a set of heuristics to reduce the search space are proposed. Details of an experimental study conducted to study the efficacy of the proposed search strategy and proposed heuristics are presented.

Finally, conclusions and future research directions are given in chapter 7.

CHAPTER 2
DECIDING THE STATUS OF ALPHA NODES

In a Gator network, an alpha node is created for every tuple variable present in a rule condition graph. An alpha node computes the tuples that match the selection predicate on its associated tuple variable in the rule condition graph. The *status* of an alpha node specifies the strategy that is used to compute the tuples matching the selection predicate.

The status of an alpha node in a Gator network can be either *stored* or *virtual.* A *stored* alpha node is a traditional type of alpha node. All the tuples matching the selection predicate are stored in it. A *virtual* alpha node, on the other hand, contains only a selection predicate and not the tuples that match the selection predicate. The tuples of a virtual alpha node are computed on demand from the base relation by applying the selection predicate. If the size of a stored alpha node is huge, an index may need to be created (on a join condition attribute) on it to speed up pattern matching. To reduce the complexity, it is assumed that a maximum of one index will be created on a stored alpha node (irrespective of the number of its join attributes). Hence, the following choices are possible for the status of an alpha node: 1. stored alpha with no index, 2. stored alpha with an index on a join attribute, and 3. virtual alpha.

Deciding the status of an alpha node in a Gator network is an interesting optimization problem. A heuristic approach was used in [14] to decide the status of an alpha node in an A-TREAT network. The following heuristic was used: an alpha memory is made virtual only if there is no selection predicate on its associated tuple variable [14]. But, it may be advantageous to create a virtual alpha in other situations also, like the following ones:

- the selection predicate is a range predicate and the selectivity of the predicate is poor,
- when the base relation has an index on the selection predicate attribute or a join predicate attribute, or
- when the update frequency of the base relation is high; here, duplication of tuples in the stored alpha node results in high update cost.

As it can be seen, the status of an alpha node is dependent not only on the selectivity of the predicate but also on other factors like the size of the base relation, availability of indexes on the base relation, availability of space for duplicating the tuples in stored alpha and so on.

Deciding the status of an alpha node is important because estimating the status properly helps to estimate the cost of a Gator network accurately which in turn helps to improve the quality of the optimal Gator network generated by an optimizer. A number of Gator networks are generated during the optimization process and the status of alpha nodes must be decided in all the generated Gator networks.

Hence, the process of deciding the status of alpha nodes must be fast. We present an approach, based on cost functions, to decide the status of alpha nodes that meets these requirements.

The proposed approach consists of two parts. **First**, cost functions have been developed to decide the status of an alpha node in any given Gator network. These cost functions essentially estimate the cost of an alpha node when it is stored and when it is virtual and chooses the status to be the one with the minimum cost. The cost of a stored alpha node is the cost of processing tokens at that node plus the cost of maintaining the tokens stored in it. The cost of a virtual alpha node is just the cost of processing tokens at that node.

During the optimization of Gator networks, many Gator networks are created and the status must be decided for all alpha nodes in each of the created Gator networks. Applying the cost functions developed above for all the alpha nodes 'as they are' might cause a lot of overhead. **Second**, a strategy has been proposed to apply the above cost functions with minimum overhead during the optimization process. Information extracted from the rule condition graph of a rule is utilized to achieve this.

2.1 The Proposed Approach

2.1.1 Part 1

The status of an alpha node, α_i, in a Gator network is dependent on:

- the frequency of arrival of tokens at the siblings of α_i,

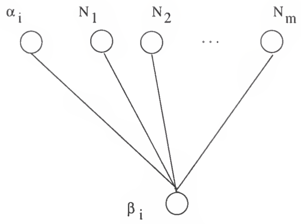


Figure 2.1. Gator sub-network

- selectivity factors of the join operations among the siblings of α_i, and
- the statistics of the base relation of α_i.

Consider the Gator subnetwork, GS_i, shown in Figure 2.1. Nodes $N_1, \ldots, N_m$ are the siblings of α_i and β_i is its parent node. $N_1, \ldots, N_m$ can either be α or β nodes. Let SP_i be the selection predicate on relation R_i, the base relation of α_i, in the rule condition. Let $fr_1, \ldots, fr_m$ be the frequency of arrival of tokens at nodes $N_1, \ldots, N_m$ respectively. When a token arrives at a node N_i, it participates in a join operation with its siblings in the order specified by the join order plan of N_i and the matching compound tuples are passed through its parent node β_i. If the status of α_i is virtual, a join operation between α_i and a token essentially requires finding the tuples of relation R_i that satisfy both the selection predicate and the join predicate. If α_i is stored, only those tuples of α_i that satisfy the join predicate need to be retrieved. The proposed method selects the best strategy for α_i which minimizes the total join processing cost.

Let T_j be the number of tuples that need to be joined to α_i due to a single token arriving at a sibling, N_j, $1 \leq j \leq m$, of α_i. If fr_i is the frequency of tokens at N_j then the frequency of tuples at α_i (tuples that need to be joined to α_i) due to the tokens arriving at N_j is given by, $f_j = fr_j * T_j$. T_j is dependent on the selectivities of the join operations in the join order plan of N_j. Let f_T be the total estimated frequency of tuples at α_i due to the tokens arriving at all of its siblings $N_1, \ldots, N_m$. f_T is given by:

$$f_T = \sum_{i=1}^{m} f_i$$

When α_i is virtual, strategies to process tokens include:

Plan 1 Scan the base relation R_i and apply the selection predicate and the join predicate on each tuple. Let C_{plan1} be the estimated cost of this plan.

Plan 2 Use an index on the selection predicate attribute (if available) on R_i to retrieve tuples satisfying the selection predicate and then apply the join predicate on the retrieved tuples. Let C_{plan2} be the estimated cost of this plan.

Plan 3 Use an index on the join predicate attribute (if available) to retrieve tuples of R_i satisfying the join predicate and then apply the selection predicate. Let $C_{join-using-index_j}$ be the estimated cost of retrieving the relevant tuples (tuples satisfying the join predicate and the selection predicate) using an index on the join predicate attribute $attrib_j$. Let $attrib_1, \ldots, attrib_m$ be the attributes of α_i that participate in a join with nodes $N_1, \ldots, N_m$ respectively. For simplicity, it is assumed that each join predicate has only one attribute of α_i. Cost functions can easily be extended for the case where a join predicate has more than one attribute of α_i. Let R_i have an index on attribute $attrib_j$ (again, the cost functions can easily be extended for the case with more than one index). Let f_j be the frequency of tuples at α_i due to N_j.

The total estimated cost of Plan3 is:

$$C_{plan3} = f_j * C_{join-using-index_j} + (f_T - f_j) * min(C_{plan1}, C_{plan2})$$

The estimated cost of processing tokens at virtual α :

$$\begin{aligned} C_{virtual} &= min(f_T * C_{plan1}, f_T * C_{plan2}, \\ &\qquad f_j * C_{join-using-index_j} + (f_T - f_j) * min(C_{plan1}, C_{plan2})) \end{aligned}$$

When α_i is stored, strategies to process tokens include:

Plan 4 scan α_i and apply join predicate on each tuple. Let C_{scan} be the estimated cost. Materializing tuples in α_i increases the update cost and consumes extra space. Let $C_{update-tuple-cost}$ be the cost to update tuples of stored α and $C_{space-tuple-cost}$ be the cost we are charging for the space occupied by the tuples of stored α .

The total estimated cost of Plan4 is:

$$C_{plan4} = F_T * C_{scan} + C_{update-tuple-cost} + C_{space-tuple-cost}$$

Plan 5 Create an index on stored alpha to retrieve tuples. To simplify the calculations, it is assumed that a maximum of one index will be maintained on an alpha node. Let C_{index_j} be the cost of retrieving tuples using an index on attribute $attrib_j$. Here, both the stored alpha tuples and the index need to be updated with each update to R_i. Let $C_{update-index-cost_j}$ be the estimated cost of updating the index on $attrib_j$ and $C_{space-index-cost_j}$ be the cost we are charging for the space occupied by the index structure created on the attribute $attrib_j$. Let $attrib_j$ be the join attribute of α_i with its sibling N_j and let f_j be the frequency of tuples at α_i due to N_j.

The estimated cost of Plan 5 with an index on the attribute $attrib_j$ is:

$$\begin{aligned} C_{plan5-indexOn_j} &= f_j * C_{index_j} + (F_t - f_j) * C_{scan} + C_{update-tuple-cost} + \\ &\quad C_{update-index-cost_j} + C_{space-tuple-cost} + C_{space-index-cost_j} \end{aligned}$$

The estimated cost of plan5:

$$C_{plan5} = min(C_{plan5-indexOn_1}, \ldots, C_{plan5-indexOn_m})$$

The estimated cost of stored α (the cost of processing tokens plus the cost of maintaining tuples that are stored in it):

$$C_{stored} = min(C_{plan4}, C_{plan5})$$

The status of α_i can be decided from $C_{virtual}$ and C_{stored}. If $C_{virtual} < C_{stored}$, the status of α_i is made virtual else it is made stored. If the status of α_i is decided to be stored, the decision of creating an index on it will be made as below: If C_{stored} is equal to C_{plan4} then no index will be created. If C_{stored} is equal to $C_{plan5-indexOn_i}$ for some attribute $attrib_i$, then an index will be created on it on attribute $attrib_i$.

2.1.2 Part 2

The above analysis shows that the status of an α node depends on the Gator network shape. Hence, its status in one network might not be appropriate in another network. Since the shape of a Gator network changes in each iteration during optimization, the cost functions also need to be applied in each iteration (after applying a local change operator and before estimating the cost). But, computing all the cost functions for all alpha nodes in each iteration might cause a lot of overhead.

Next, we propose a method to compute $C_{virtual}$ and C_{stored} with minimum overhead during optimization. This method uses information from the condition graph of a rule. The condition graph gives information about the join operations between different relations in the rule condition.

The method is based on the following observations about the cost functions:

1. The attributes of α_i in a join condition with any of its sibling nodes N_i in any Gator network is always a subset of the join attributes of R_i as indicated by the rule condition graph.

2. Cost functions C_{plan1}, C_{plan2}, C_{scan}, $C_{update-tuple-cost}$ and $C_{space-tuple-cost}$ are dependent only on the statistics of R_i and SP_i. C_{index_j}, $C_{join-using-index_j}$,

$C_{update-index-cost_j}$ and $C_{space-index-cost_j}$ are dependent only on the join attributes of α_i with its sibling N_j (refer to the subnetwork in Figure 2.1) and not on any other properties of N_j.

From observations 1 and 2, it follows that by pre-computing the cost functions (that are dependent only on the join attributes of α_i and are independent of the actual siblings of α_i) for all the possible join attributes of α_i and storing them in a table, they don't need to be computed again in each generated Gator network during optimization. Only the join attributes of α_i and the f_i's of siblings need to be computed from the changed or new Gator network. And, the values of cost functions for the join attributes in the changed Gator network can be accessed from the table. The size of the cost table is linear to the number of the join attributes of R_i. If k is the number of edges between R_i and the other nodes in the rule condition graph, the cost table contains $4k + 5$ entries.

2.1.3 Part 3

A brief sketch of the method that is based on the above ideas is presented next. It computes $C_{virtual}$ and C_{stored} with minimum overhead during optimization.

Stage 1: Pre-processing: For each α node, calculate all the cost functions that are dependent on only the join attributes, and store them in its cost table.

```
For each alpha node, $\alpha_i$, in the rule do
    1. Calculate $C_{plan1}$, $C_{plan2}$, $C_{scan}$, $C_{update-tuple-cost}$ and $C_{space-tuple-cost}$ and store
       them in the cost table of $\alpha_i$.
```

2. Extract the number of edges between the base relation of α_i and the other relations (in the rule condition) from the condition graph of the rule. For each join attribute, $attrib_j$, of the base relation of α_i, calculate C_{index_j}, $C_{join-using-index_j}$, $C_{update-index-cost_j}$ and $C_{space-index-cost_j}$ and store them in the cost table of α_i.

end

Stage 2: Deciding the status of α nodes during optimization: Randomized algorithms are being used to optimize Gator networks. Refer to chapter 4 for more details on the Gator network optimizer. The status of alpha nodes is decided in each Gator network that is generated during the optimization process. The decision procedure is given below.

For each α_i in the rule do

1. Calculate $f_1, \ldots, f_m$ for the siblings $N_1, \ldots, N_m$ of α_i.(f_i's are calculated as part of estimating the cost of new Gator network.)
2. Find the join attributes of α_i with N_i's. The values of C_{index_i}, $C_{update-index-cost_i}$, $C_{join-using-index_i}$ and $C_{space-index-cost_i}$ for these attributes and the values of C_{plan1}, C_{plan2}, C_{scan}, $C_{update-tuple-cost}$ and $C_{space-tuple-cost}$ can be obtained from the cost table of α_i.
3. Compute C_{stored} and $C_{virtual}$ using the frequencies and cost functions gathered in steps 1 and 2. The status of α_i is made virtual if $C_{virtual} < C_{stored}$ else it is made stored.

4. If the status of α_i is decided to be stored, then the decision of creating an index on it will be made as below: If C_{stored} is equal to C_{plan4}, then no index will be created. If C_{stored} is equal to $C_{plan5-indexOn_i}$ for some attribute $attrib_i$, then an index will be created on it on attribute $attrib_i$.

end

2.2 Cost Functions

The cost estimates of various plans that were utilized in deciding the status of an alpha node are given below. Please refer to chapter 3 for details on the parameters that are used in these cost formulae.

The value of C_{plan1} includes the I/O cost for relation scan plus the CPU cost to apply the selection predicate and the CPU cost to apply the join predicate. The formula for C_{plan1} is thus:

$$C_{plan1} = I/O_{weight} \times Pages(R) + 2 \times CPU_{weight} \times Card(R)$$

The value of C_{plan2} includes the I/O cost to retrieve relevant tuples (using an index on the select ion predicate attribute) plus the CPU cost to apply the join predicate and the CPU cost to access index pages. The formula for C_{plan2} is thus:

$$\begin{aligned} C_{plan2} &= I/O_{weight} \times Yao(Card(R), Pages(R), \lceil \frac{Card(R)}{Card(R.attr)} \rceil) + \\ &\quad CPU_{weight} \times (\frac{Card(R)}{Card(R.attr)} + \lceil \log_{fanout} Card(R) \rceil) \end{aligned}$$

The value of $C_{join-using-index_i}$ includes the I/O cost to retrieve relevant tuples (using an index on the join predicate attribute *attrib*$_i$) plus the CPU cost to apply the selection predicate and the CPU cost to access index pages. The formula for $C_{join-using-index_i}$ is thus:

$$\begin{aligned} C_{join-using-index_i} &= I/O_{weight} \times Yao(Card(R), Pages(R), \lceil \frac{Card(R)}{Card(R.attr)} \rceil) + \\ & CPU_{weight} \times (\frac{Card(R)}{Card(R.attr)} + \lceil \log_{fanout} Card(R) \rceil) \end{aligned}$$

C_{plan4} is the cost of stored alpha node without any index on it. The formula for C_{plan4} is given by:

$$C_{plan4} = F_T \times C_{scan} + C_{update\_tuple\_cost}$$

where C_{scan} is the cost of scanning the tuples of stored alpha node

$$C_{scan} = CPU_{weight} \times Card(\alpha) + I/O_{weight} \times Pages(\alpha)$$

and $C_{update\_tuple\_cost}$ is the cost to update the tuples of stored alpha node.

$$\begin{aligned} C_{update\_tuple\_cost} &= F_i \times [CPU_{weight} + 2 * I/O_{weight}] + \\ & F_d \times [CPU_{weight} \times Card(\alpha) + I/O_{weight} \times (pages(\alpha) + 1)] \end{aligned}$$

$C_{plan5\_indexOn_i}$ is the cost of stored alpha node that has an index on the join attribute. The formula for $C_{plan5\_indexOn_i}$ is given by:

$$\begin{aligned} C_{plan5\_indexOn_i} &= F_i \times C_{index_i} + (F_T - F_i) \times C_{scan} + C_{update\_tuple\_cost} + \\ & \qquad C_{update\_index\_cost_i} + C_{space\_tuple\_cost} + C_{space\_index\_cost_i} \end{aligned}$$

where C_{index_i} is the cost of processing joins at stored alpha node using an index on the join attribute

$$\begin{aligned} C_{index_i} &= I/O_{weight} \times Yao(Card(\alpha), Pages(\alpha), \lceil \frac{Card(R)}{Card(R.attr)} \rceil) + \\ & \qquad CPU_{weight} \times (\frac{Card(R)}{Card(R.attr)} + \lceil \log_{fanout} Card(\alpha) \rceil) \end{aligned}$$

and $C_{update\_index\_cost_i}$ is the total cost of updating the index pages in main memory.

$$C_{update\_index\_cost_i} = (F_i + F_d) \times CPU_{weight} \times \lceil \log_{fanout} Card(\alpha) \rceil$$

2.3 Conclusion

The status of an alpha node specifies the strategy that is used to compute the tuples matching the selection predicate that is associated with it. Possible choices for the status of an alpha node are: Stored with no index, Stored with an index on a join attribute and Virtual. This chapter proposed a strategy that can decide the status of alpha nodes in a Gator network, with minimum overhead, during the Gator network optimization process. The technique is based on cost functions and it utilizes information extracted from the condition graph of a rule. The highlight of this approach is that it computes most of the cost functions (that are needed to decide the status of an alpha node) in advance, thereby, reducing the overhead during the optimization process. The strategy has been implemented as part of the Ariel active relational DBMS.

The next chapter presents cost functions to estimate the cost of a Gator network. Chapter 4 discusses the Gator network optimization problem.

CHAPTER 3
COST FUNCTIONS

Gator networks are general tree structures. There are many possible Gator networks that can do pattern matching for a given rule. This chapter presents the cost model that has been developed to estimate the cost of a Gator network so that an optimizer can compare different Gator networks and choose an optimal one. An optimal Gator network for a rule is the one with the lowest cost.

The cost functions are based on the statistical information available in system catalogs such as relation cardinalities and attribute cardinalities. In the Gator network cost model, the cost functions used to estimate selection and join predicate selectivities are similar to the ones used in system R [44]. An important parameter used in Gator network cost model which is different from the ones used in query optimizer cost estimates is the *update frequency distribution* of relations. The relative update frequency of a relation R gives an estimate of the fraction of times a trigger condition is tested due to (updates to) that relation relative to the other relations in the trigger condition. The update frequencies of relations can have major impact on the shape of an optimal Gator network. It is assumed that the update frequencies of relations are also maintained in system catalogs.

Two cost models have been developed. In both the models, both the CPU cost and the I/O cost are taken into account. Model 1 assumes the availability of large buffer space. Model 2 assumes that minimum buffering is available for all operations. In both the models, only B^+-tree indexes are assumed to be available on relations and discrimination network nodes. During token propagation, only nested-loop join (possibly with an index on the inner node or table) is allowed to find matches. It is assumed that heap organization is used to store tuples in relations and discrimination network nodes. Also, no pipelining of results is assumed between the operations. The cost functions presented here are an extension of the ones given in [15].

The cost of a Gator network is the sum of the following:

1. The cost of performing selections and joins (i.e. testing the rule condition) due to tokens entering the discrimination network.
2. The cost of maintaining the nodes of the discrimination network. The data materialized in the nodes has to be kept consistent with the data in the database and hence it needs to be updated whenever changes happen to the database.
3. Space cost. There is a space cost associated with materializing the alpha and beta nodes. For simplicity, the space cost is set to zero for all the discrimination network nodes.

The cost model given below assumes that buffer space is limited, charging an amount I/O_{weight} for each I/O. However, the effect of having large buffer space can be approximated by setting I/O_{weight} to zero or some very small value. The small and large buffer space cost models are referred to as CM1 and CM2, respectively.

The cost formulae for a Gator network are defined recursively. The cost of a node in a Gator network is defined as the sum of the costs of all the nodes in the subnetwork rooted at that node. The parameters that are used in the cost functions are given next. The cost functions for the alpha, beta and the P-node are given after that.

Name	Description
CPU_{weight}	The relative weight of a CPU operation
I/O_{weight}	The relative weight of an I/O operation
$R(\alpha)$	The base relation of the α-node, α.
N	Any node in a discrimination network: α, β or a P-node.
$Sel(\alpha)$	The selectivity factor of the selection predicate associated with an α-memory node, α.
$F_i(R)$:	The insert frequency of relation R relative to other relations.
$F_d(R)$:	The delete frequency of relation R relative to other relations.
$F_i(N)$:	The relative insert frequency of a Gator network node, N
$F_d(N)$:	The relative delete frequency of a Gator network node, N
$pageSize$	The page size in bytes.
$tupleSize(N)$	The size of a tuple in node N in bytes.
$tuplesPerPage(N)$	Number of tuples of node N per page.
$Pages(N)$:	The number of pages occupied by a node N.

$Card(N)$:	Cardinality of N.
$Card(N.attr)$:	Cardinality of attribute *attr* in N.
$Card(R)$:	Cardinality of relation R.
$Card(R.attr)$:	Cardinality of attribute *attr* in R.
$JSF(N_i, N_j)$:	The selectivity factor of the join condition between the nodes N_i and N_j .
$leaves(\beta)$:	Indicates the leaf nodes of the tree rooted at β in a discrimination network.
$fanout$:	Fanout of a node in a B^+-tree. Cost functions involving indexes are given only for B^+-trees.

3.1 Cost Functions for an α Node

An alpha node holds all the tuples of its base relation that satisfy the selection condition that is associated with it. The cardinality, insertion and deletion frequencies of an α node are estimated as below:

Cardinality, $\text{Card}(\alpha) = \text{Card}(\text{R}(\alpha)) \times \text{Sel}(\alpha)$

Insert Frequency, $F_i(\alpha) = F_i(R(\alpha)) \times Sel(\alpha)$

Delete Frequency, $F_d(\alpha) = F_d(R(\alpha)) \times Sel(\alpha)$

The cost of an alpha node is given by the cost of maintaining the tuples that are stored in it. The insert cost of an α node, $C_i(\alpha)$, is the cost of inserting a tuple into it. The delete cost, $C_d(\alpha)$, is the cost of deleting a tuple from the α node.

$$Cost(\alpha) = F_i(\alpha) \times C_i(\alpha) + F_d(\alpha) \times C_d(\alpha)$$

As explained in chapter 2, there are three choices for the status of an α node: 1. Stored, 2. Virtual, and 3. Stored with an index on a join condition attribute. The cost functions for C_i and C_d for the three cases are given below.

3.1.1 Insertion Cost of an α Node

Stored alpha node

The tuples of an alpha node are assumed to be stored in a heap. Hence, the insertion cost is the cost of updating the page in which the new tuples are going to be stored.

$$C_i(\alpha) = CPU_{weight} + 2 \times I/O_{weight}$$

Virtual alpha node

Since a virtual-alpha node does not store any tuples, the insertion cost is zero.

$$C_i(\alpha) = 0$$

Stored alpha node with an index on a join condition attribute

Here, in addition to inserting the tuple in the alpha node, the index node pages also need to be updated. It is assumed that only the root nodes of the indexes are available in main memory.

$$C_i(\alpha) = CPU_{weight} + 2 \times I/O_{weight} + \lceil \log_{fanout} Card(\alpha) \rceil \times CPU_{weight}$$

3.1.2 Deletion Cost of an α Node

Stored alpha node

Since the tuples are stored in a heap, a scan of all the pages in the node needs to be done to delete the tuples. Assuming that all the tuples that need to be deleted are in one page, we get the following formula:

$$C_d(\alpha) = CPU_{weight} \times Card(\alpha) + I/O_{weight} \times (Page(\alpha) + 1)$$

Virtual alpha node

Virtual alpha node do not store any tuples and hence the deletion cost for a virtual alpha node is zero.

$$C_d(\alpha) = 0$$

Stored alpha node with an index on a join condition attribute

Here, in addition to deleting the tuples in the alpha node, the index node pages also need to be maintained.

$$\begin{aligned} C_d(\alpha) &= CPU_{weight} \times Card(\alpha) + I/O_{weight} \times (Page(\alpha) + 1) + \\ &\quad \lceil \log_{fanout} Card(\alpha) \rceil \times CPU_{weight} \end{aligned}$$

3.2 Cost Functions for a β Node

A beta node in a Gator network can have two or more child nodes. A token arriving at a child node is joined with all the other child nodes of the beta node. Depending upon the type of the token (insert/delete), matching tokens are either inserted in or deleted from the beta node and a copy of them is propagated to the parent of the beta node.

Cost functions to estimate the cardinality and the insert and delete frequencies of a beta node are presented next. Cost functions to estimate the cost of a beta node are presented after that.

3.2.1 Estimating the Cardinality of a β Node

The size and the insert and delete frequencies of a beta node are independent of the shape of the subnetwork rooted at that beta node. They are dependent only on the leaf nodes of the subnetwork rooted at the beta node.

Let $\prod_S(\beta) = \prod_{\alpha \in leaves(\beta)} Card(R(\alpha))$,

$\prod_\sigma(\beta) = \prod_{\alpha \in leaves(\beta)} Sel(\alpha)$ and

$\prod_\psi(\beta) = \prod(JSF(\alpha_i, \alpha_j))$, where α_i, $\alpha_j \in leaves(\beta)$ and $\exists edge(Rel(\alpha_i, Rel(\alpha_j))$ in the rule condition graph.

$\prod_S(\beta)$ represents the estimated size of the cartesian product of the base relations of the leaf nodes of the beta node. $\prod_\sigma(\beta)$ represents the product of the selectivities of the selection conditions associated with the alpha nodes in $leaves(\beta)$. $\prod_\psi(\beta)$ represents the product of the selectivity factors of the join conditions between the leaf nodes of the beta node.

The cardinality of a β node is defined as below:

$$Card(\beta) = \prod_{\sigma} (\beta) \times \prod_{\psi} (\beta) \times \prod_{S} (\beta)$$

3.2.2 Estimating the Insert and Delete Frequencies of a β Node

Let $TokenGenCount(N, \beta)$ represent the number of tokens generated at a β-node, β, due to a single token arriving at a child node, N, of β.

The insert and delete frequencies of a beta node are defined as follows:

$$F_i(\beta) = \sum_{N \in children(\beta)} F_i(N) \times TokenGenCount(N, \beta)$$

$$F_d(\beta) = \sum_{N \in children(\beta)} F_d(N) \times TokenGenCount(N, \beta)$$

$JoinSizeAndCost(N, \beta)$ estimates the join result size and the cost of performing a sequence of join operations when a token arrives at a child node N of the beta node. Details about JoinSizeAndCost are given later. $TokenGenCount(N, \beta)$ is expressed in terms of $JoinSizeAndCost(N, \beta)$ below.

```
$TokenGenCount(N, \beta)$
{
  (size, cost) = $JoinSizeAndCost(N, \beta)$;
  return(size)
}
```

3.2.3 Estimating the Cost of a β Node

The cost of a beta node is the sum of the following:

1. The cost of the child nodes of the β node.

2. The cost of performing joins for tokens fed into the child nodes of the β node.

3. The cost associated with maintaining (updating) the β node.

A formula for the cost of a β node is thus:

$$Cost(\beta) = LocalCost(\beta) + \sum_{N \in children(\beta)} Cost(N)$$

where

$$\begin{aligned} LocalCost(\beta) \quad = \quad & \sum_{N \in children(\beta)} \{F_i(N) \times PerChildInsCost(N, \beta) + \\ & F_d(N) \times PerChildDelCost(N, \beta)\} \end{aligned}$$

$PerChildInsCost(N, \beta)$ represents the cost of processing a "+" token arriving at a child node, N, of the beta node. $PerChildInsCost(N, \beta)$ consists of two components: 1. The cost of performing a multi-way join on a tuple arriving at a child N of the β node. Following the join order plan associated with the node N, a sequence of two-way joins with each of the siblings of N is performed. 2. The cost of updating the β node with the resulting compound tokens (if any). $JoinSizeAndCost(N, \beta)$ estimates the cost of performing join operations between the child nodes of β and a token arriving at node N.

```
$PerChildInsCost(N, \beta)$ {
    (size, cost) = $JoinSizeAndCost(N, \beta)$
    insertCost = $\lceil \frac{size}{tuplesPerPage(\beta)} \rceil \times 2 \times I/O_{weight}$ + size $\times CPU_{weight}$
    return (cost + insertCost)
}
```

PerChildDelCost(N, β) represents the cost of processing a "–" token arriving at a child N of the beta node. The cost function *PerChildDelCost*(N, β), similar to *PerChildInsCost*(N, β), has two components: 1. The cost of performing a multi-way join on a tuple arriving at a child N of the β node. 2. The cost of deleting the resulting compound tokens (if any) from the β node. *PerChildDelCost*(N, β) is expressed in terms of *JoinSizeAndCost*(N, β) below.

```
$PerChildDelCost(N, \beta)$ {
    (size, cost) = $JoinSizeAndCost(N, \beta)$
    deleteCost = (Yao(Card($\beta$),Pages($\beta$),TRSize) + Pages($\beta$)) $\times I/O_{weight}$ +
                 Card($\beta$) $\times CPU_{weight}$
    return (cost + deleteCost)
}
```

The Yao(n,m,k) function estimates the number of pages that would be touched when k tuples are randomly selected within relations/nodes that occupy m pages, each page containing n/m records.

Estimating JoinSizeAndCost(N, β): Every child node of a β is associated with a join order plan which gives the order in which a token arriving at that node is to be joined with the siblings of that node. $JoinSizeAndCost(N, \beta)$ estimates the join size and the cost of performing join operations (following the join order plan of N) with the child nodes of the beta node when a token arrives at node N. An intermediate relation is created for the results of the join and it is used to join with the other child nodes of the beta node.

Let TR represent the intermediate relation formed during the join process, $TRSize$ represent the estimated cardinality of TR and Card(TR.joinAttr) represent the estimated cardinality of the join attribute in TR.

```
$JoinSizeAndCost(N, \beta)$ {
  TRSize = 1
  TempCost = 0
  for each node $n$ in the Join Order plan of N
  {
    TempCost = TempCost + $TwoWayJoinCost(TRSize, n)$
    TRSize = TRSize $\times$ Card(n) / max(Card(TR.joinAttr) , Card(n.joinAttr))
  }
```

```
    return (TRSize, TempCost)
}
```

Estimating TwoWayJoinCost(TRSize, N): $TwoWayJoinCost(TRSize, N)$ estimates the cost of performing a join operation between an intermediate result TR of size $TRSize$ and a node N. The node N can be a stored alpha node (with or without an index on the join condition attribute), a virtual alpha node (with or without an index on the join condition attribute on the base relation) or a beta node. The cost functions for the various cases are given below.

1. Stored alpha node with no index on the join condition attribute

$$TwoWayJoinCost(TRSize, n) =$$
$$Pages(n) \times I/O_{weight} + TRSize \times Card(n) \times CPU_{weight}$$

2. Stored alpha node with an index on a join condition attribute

$$TwoWayJoinCost(TRSize, n) =$$
$$TRSize \times Yao(Card(n), Pages(n), \lceil \frac{card(n)}{Card(n.joinAttr)} \rceil) \times I/O_{weight} +$$
$$TRSize \times \lceil \frac{card(n)}{Card(n.joinAttr)} \rceil \times CPU_{weight} +$$
$$TRSize \times \lceil \log_{fanout} Card(n) \rceil \times CPU_{weight}$$

3. Virtual alpha node with no index on the join condition attribute on the base relation

$$TwoWayJoinCost(TRSize, n) =$$
$$Pages(n) \times I/O_{weight} + TRSize \times Card(n) \times CPU_{weight}$$

4. Virtual alpha node with an index on the join condition attribute on the base relation

$$TwoWayJoinCost(TRSize, n) =$$
$$TRSize \times Yao(Card(n), Pages(n), \lceil \frac{card(n)}{Card(n.joinAttr)} \rceil) \times I/O_{weight} +$$
$$TRSize \times \lceil \frac{card(n)}{Card(n.joinAttr)} \rceil \times CPU_{weight} +$$
$$TRSize \times \lceil \log_{fanout} Card(n) \rceil \times CPU_{weight}$$

5. Beta node

$$TwoWayJoinCost(TRSize, n) =$$
$$Pages(n) \times I/O_{weight} + TRSize \times Card(n) \times CPU_{weight}$$

The cardinality of a join attribute $jAttr$ in the temporary result TR is estimated in the following way [6]: Let $jAttr$ be the attribute of a relation R and let $n = Card(R)$ and $b = Card(R.jAttr)$. Assuming uniform distribution of values, independence of value distribution in different columns, and random selection of values for the join attribute in TR from the original relation, estimation of $Card(TR.jAttr)$ can be reduced to the following statistical problem: Given n objects uniformly distributed over b colors, how many different colors are selected if one randomly selects $t = \text{Card(TR)}$ objects? Bernstein et al. [2] give an approximation of this value, shown here as the function $Estimate(n, b, t)$:

$$\begin{aligned} Estimate(n, b, t) &= t, \text{ for } t < b/2 \\ &= (t + b)/3, \text{ for } b/2 \leq t < 2b \\ &= b, \text{ for } t \geq 2b \end{aligned}$$

This function is used internally to estimate the cardinality of a join attribute in a temporary result. The same approach is used to estimate the cardinalities of attributes in the α and β nodes also (except the ones that participate in a selection condition).

3.3 Cost Functions for the P-node

The cost functions for the P-node are slightly different from those of the beta node. During a transaction, all the tuples matching a rule condition are stored in the P-node of that rule. The contents of the P-node are not persistent. The tuples accumulated in the P-node during a transaction are deleted at the end of that transaction, after the rule is activated. Because of this, the size of the P-node tends to be small and it is assumed that the contents of the P-node are resident in main memory.

The cost of a P-node is given by:

$$Cost(P) = LocalCost(P) + \sum_{N \in children(\beta)} cost(N)$$

where

$$\begin{aligned} LocalCost(P) \quad = \quad \sum_{N \in children(P)} & \{F_i(N) \times PerChildInsCost(N, P) + \\ & F_d(N) \times PerChildDelCost(N, P)\} \end{aligned}$$

The cost functions for $PerChildInsCost(N, P)$ and $PerChildDelCost(N, P)$ are given below:

```
$PerChildInsCost(N, \beta)$ {
  (size, cost) = $JoinSizeAndCost(N, \beta)$
  insertCost = size $\times CPU_{weight}$
  return (cost + insertCost)
}
```

Since the tuples of the P-node are assumed to be in main memory, there is no I/O cost involved in inserting new tuples in the P-node. Also, for the same reason, there is no I/O cost involved in deleting tuples from the P-node. $PerChildDelCost(N, P)$ is given next.

```
$PerChildDelCost(N, \beta)$ {
  return $(CPU_{weight})$
}
```

3.4 Conclusion

This chapter discussed the problem of estimating the cost of a Gator network of a rule, given information about the structure of the rule, database size, attribute cardinalities and update frequency distribution. Two cost models were developed based on the amount of buffer space that is available for trigger processing. In both the models, both the CPU cost and the I/O cost were taken into account. The cost of a Gator network is the sum of the cost of performing selections and joins due to tokens entering the Gator network and the cost of maintaining the nodes of the Gator network (the Gator network nodes may need to be updated whenever changes happen to the database). The cost functions used to estimate the selectivities of predicates are similar to the ones in [44].

The Gator network optimizer uses this cost model to compare different Gator networks during the optimization process. The details of the Gator network optimizer are given next.

CHAPTER 4
GATOR NETWORK OPTIMIZATION

Gator networks are general tree structures. There exist many Gator networks that can do pattern matching for a given rule. Hence, there is a need for a Gator network optimizer to find an optimal Gator network for a rule. An optimal Gator network of a rule is a Gator network that can do pattern matching faster than any other Gator network of that rule.

Optimizing Gator networks is a combinatorial optimization problem. Randomized algorithms have successfully been applied in various hard combinatorial problems [1, 59]. Swami [57] and Ioannidis [24, 22, 23, 28] applied randomized algorithms for optimizing large join queries. This work motivated us to use a randomized algorithms-based approach for optimizing Gator networks. Also, a simulation study conducted in [17, 39] showed that a randomized algorithms-based approach is superior to the dynamic programming approach for the Gator network optimization problem. The dynamic programming approach broke down for rule conditions with eight or more tuple variables. Our goal in this work is to be able to optimize rules having up to 15 tuple variables in a few minutes. (This is the same as the maximum number of relations allowed in an SQL SELECT statement in DB2 and Sybase SQL Server).

The number of Gator networks for a rule condition is extremely large compared to the number of Rete networks (or query plans for an equivalent query). The following two theorems give an estimate on the number of Rete and Gator networks for a given rule.

Theorem 1 *For a rule having a condition graph with* N *joins, the number of different Rete networks is given by* $\binom{2N}{N}N!$.

Proof: The different number of binary trees that can be created with $N+1$ leaf nodes is given by $\binom{2N}{N}\frac{1}{N+1}$ [30]. The number of ways that $(N+1)$ tuple variables can be associated with the leaf nodes in each of these binary trees is given by $(N+1)!$. A Rete network is essentially a binary tree with materialized internal nodes. Hence, the number of Rete networks is given by $\binom{2N}{N}N!$.

Theorem 2 *For a rule having a condition graph with* N *joins, the number of different Gator networks is upper bounded by* $2^{N-1}\binom{2N}{N}N!$.

Proof: Consider the sub-network (SN1) shown by marked lines in the Gator network, GN1, in Figure 4.1. A, B, C and D represent alpha nodes in the figure. SN1 has a beta node and three alpha nodes. The networks SN2 and SN3 have the same leaf nodes as that of SN1. The join order plans of the leaf nodes in SN2 and SN3 are the same as those of the corresponding ones in SN1. It is important to note that the internal beta nodes in SN2 and SN3 are not materialized.

The following observations can be made about the three sub-networks SN1, SN2 and SN3:

1. The pattern matching costs of the three sub-networks SN1, SN2 and SN3 are identical. This is because of the fact that the join order plans of the nodes in the three subnetworks are identical.

2. The state information maintained in the three subnetworks is also identical.

From the above observations, it can be said that the three sub-networks are just different representations of one another and are equivalent. Since SN1 and SN2 are equivalent, the Gator networks GN1 and GN2 are also equivalent (SN1 is replaced by SN2 in GN2). By extending this argument, it can be said that any Gator network can be converted into an equivalent binary tree network with a proper choice of materialized and non-materialized beta nodes. Also, by taking a binary tree network and by assigning the materialized/non-materialized status to its beta nodes, we can generate different Gator networks. It may be noticed that some Gator networks (which have at least one beta node that has more than two child nodes) are equivalent to more than one binary tree-shaped networks. For example, the Gator network GN1 is equivalent to six binary tree-shaped networks and GN2 is one of them. Hence, by enumerating all possible binary tree networks with all possible assignments of materialized/non-materialized status to its beta nodes, we can find an upper bound on the number of Gator networks.

The number of different binary trees that can be constructed with $N+1$ leaf nodes is given by $\binom{2N}{N}\frac{1}{N+1}$ [30]. The number of ways that $(N+1)$ tuple variables can be associated with the leaf nodes in each of these binary trees is given by $(N+1)!$. Also, there are N internal nodes in a binary tree with $N+1$ leaf nodes. For each

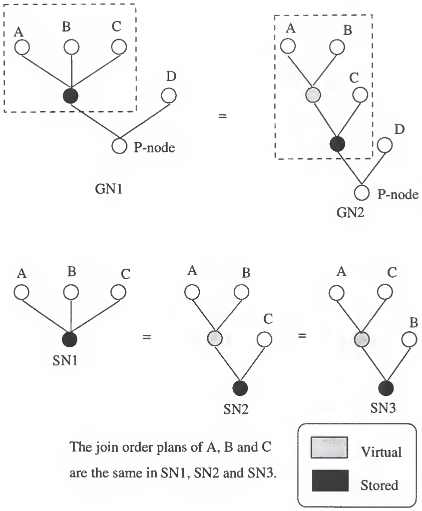

A
B
C
D
P-node
GN1
=
A
B
C
D
P-node
GN2
A
B
C
SN1
=
A
B
C
SN2
=
A
C
B
SN3
The join order plans of A, B and C are the same in SN1, SN2 and SN3.
Virtual
Stored

Figure 4.1. Equivalence of Gator networks

of the internal nodes (except the P-node, which is never materialized), based on the decision whether to materialize it or not, we can generate a different Gator network. Hence, the upper bound on the number of Gator networks is given by $2^{N-1}\binom{2N}{N}N!$.

To cope with the large search space for Gator networks, the Gator network optimizer uses a randomized search technique. Three randomized algorithms were explored: Iterative Improvement (II), Simulated Annealing (SA), and Two-Phase Optimization (TPO). General terminology of randomized algorithms is given next. Details of the algorithms are given after that.

4.1 Randomized Algorithms for Optimizing Gator Networks

Each solution to a combinatorial optimization problem can be viewed as a *state* in a state space. Each state in the state space is associated with a cost, calculated by using a problem-specific cost function. The aim of an optimization algorithm is to find a state with the lowest cost in the state space. A *move* or a *local change operator* is an operation applied to a state to obtain another state. All the states that can be reached from a state in one move are called the *neighbors* of that state. A state is called a *local minimum* if the cost of the state is less than that of all its neighbor states. A *global minimum* is the state with the lowest cost in the state space. A move is called a *downhill move* if the cost of the new state obtained by applying a move is less than that of the current state; or if the cost of the new state is higher, it is called an *uphill move.* A *plateau* consists of adjacent states with the same cost.

```
procedure II()
{
   minState = random state;
   while not (stopping_condition())
   {
      S = random state
      while not(local_minimum(S))
      {
         S' = random state in neighbors(S)
         if cost(S') < cost(S)
             S = S'
      }
      if cost(S) < cost(minState)
          minState = S
   }
   return(minState)
}
```

Figure 4.2. Iterative Improvement

4.2 Generic Algorithms

4.2.1 Iterative Improvement

Iterative Improvement [36, 1] is a local search algorithm that comes under the class of heuristic or approximation algorithms.

The generic algorithm is shown in Figure 4.2. In each iteration of the outer loop, a random state is generated and a local search is initiated with the generated random state as the start state. The local search process (the inner loop) starts at a given state and applies downhill moves repeatedly until a local minimum is reached. The number of iterations of the outer loop (i.e. the number of local minimum states examined) is controlled by the *stopping criterion*. The output of II is the local minimum state with the lowest cost.

```
S = initialize();
T = initialTemp();

repeat {
    repeat {
        newS = move(S);
        delta = cost(newS) - cost(S);
        if (delta <= 0)
            S = newS;
        if (delta > 0)
            S = newS with probability exp(-delta/T);
    } until (inner-loop criterion is satisfied);

    T = reduceTemp(T);
} until(system has frozen);

return (minS);
```

Figure 4.3. Simulated Annealing

4.2.2 Simulated Annealing

Simulated Annealing [29, 1, 59] is a generalization of local search algorithms. A local search algorithm accepts only down-hill moves whereas SA accepts both downhill and uphill moves. In SA, the probability of accepting uphill moves is controlled by a parameter called *temperature*. This feature of accepting both uphill and downhill moves helps SA avoid becoming trapped in high cost local optimal solutions.

SA has been proposed by Kirkpatrick et al. [29] and is based on the annealing of solids. The generic algorithm is given in Figure 4.3. The algorithm starts with an initial state and an initial temperature. It performs a number of local searches called *stages*. Each stage is run with a constant temperature. In all the stages, downhill moves are always accepted whereas uphill moves are accepted with a probability of

$e^{(-\triangle c/t)}$, where $\triangle C$ is the difference in the cost of a new state and the original state and t is the temperature during that stage. Thus, the probability of accepting higher cost states is higher at higher temperatures. Also, the probability is higher when $\triangle C$ is less. Each stage ends when a terminating condition is satisfied. After every stage, the temperature is lowered according to a pre-defined function *reduceTemp* and a new stage is started with the new temperature. The algorithm terminates when the freezing criterion is met and outputs the lowest cost state visited.

4.2.3 Two Phase Optimization

Two Phase Optimization (2PO) was proposed in [22, 28]. 2PO is a combination of II and SA. In the first phase of 2PO, II is run for a small period of time. In the second phase, SA is run with a low initial temperature. The output of II is given as the initial state to SA. The output of SA is returned as the output of 2PO. It has been shown in [22, 23, 28] that, when the state space satisfies certain properties, 2PO has the potential to perform better than II and SA.

A *condition graph node set* of a Gator network node N, CGS(N), is defined to be the set of nodes in the condition graph corresponding to the leaf alpha nodes of N. Two Gator network nodes N1 and N2 are *connected* if there is a rule condition graph edge between an element of CGS(N1) and CGS(N2).

The details of the problem specific parameters are given next.

4.3 Problem Specific Parameters

4.3.1 State Space

The goal of the rule condition optimization problem is to find a Gator network shape that can minimize the rule condition testing time for a rule. The state space of the Gator network optimization problem for a rule consists of all the complete Gator networks that can be constructed for that rule. The following heuristic is used to constrain the state space: a beta node is never created with child nodes that do not have a join condition between them in the rule condition. The status of alpha nodes in a Gator network is decided based on the analysis given in chapter 2. If a beta node has more than two child nodes, the join order for those nodes is generated based on the following heuristic: the join order is the order obtained by sorting its sibling nodes according to their size.

4.3.2 Neighbors Function

The local change operators that were chosen for the Gator network optimization problem are given below. They are also illustrated in Figure 4.4.

- **Create-Beta**: Create-Beta adds a new beta node to the discrimination network. A beta node, β_1, is picked randomly from the set of beta nodes having more than two child nodes. Two connected child nodes of β_1 are picked randomly and are made the child nodes of a new Beta node, β_2. β_2 is then made a child node of β_1.

- **Kill-Beta**: Kill-Beta removes a randomly picked beta node, β_1, from the discrimination network. The child nodes of β_1 are made the child nodes of the parent node of β_1.

- **Merge-Sibling**: Merge-Sibling merges two sibling nodes of a randomly picked beta node. A beta node having more than two child nodes is randomly selected. Two connected siblings of this beta node are randomly picked and one of them is made a child node of the other. The node to which a child node is added must be a beta node.

4.3.3 Cost Functions

Cost functions are developed to estimate the cost of a Gator network relative to other Gator networks for a trigger. Two cost models are developed, one for plentiful-buffer environment (CM1) and another for low-buffer environment (CM2). Details of cost functions were given in chapter 3.

4.4 Generating a Random Gator Network

The algorithm for creating a random Gator network is given below:

1. Let the rule condition graph have n tuple variable nodes. Create an alpha node for each of the tuple variable nodes in the graph and insert them in a list.

2. Repeat the following until there is only one node in the list:

   (a) Select a node, N_i, randomly from the list.

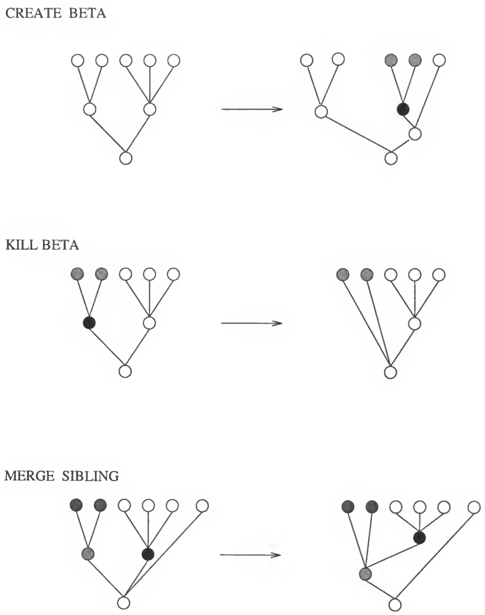


Figure 4.4. Example application of local change operators

parameter	value
stopping condition	same time as that of TPO
local minimum	r-local minimum
next state	random neighbor

Figure 4.5. Parameters for II

parameter	value
initial state	random state
initial temp	2 * cost(initial state)
temp reduction	0.95^*T_{old}
frozen	same as that of the SA phase of TPO

Figure 4.6. Parameters for SA

(b) Find all the nodes in the list that are connected to N_i. Let N_{list} be the list of nodes that are connected to N_i. Let NL be the number of nodes in N_{list}.

(c) Select a number, K, randomly between 1 and NL. Select K nodes from N_{list} and create a beta node with these K nodes and the node N_i as children.

(d) Delete the above selected K nodes and N_i from the list. Add the created beta node to the list.

3. When there is one node left in the list, it gives the complete Gator network for the trigger.

parameter	value
stopping condition (II phase)	20 local optimizations
initial state	best of II ($best_{II}$)
initial temperature (T_0)	0.5*cost($best_{II}$), if cost($best_{II}$) $<$ 20000 0.05*cost($best_I I$), otherwise
equilibrium	Number of edges in the rule condition graph
temp reduction	0.95*T_{old}
frozen	temp $< T_0/1000$ and best state unchanged for 5 stages Or total time >= (40*number of relations in rule condition) seconds

Figure 4.7. Parameters for TPO

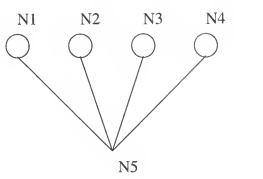
4.5 Optimizer Tuning

The parameters used in this study for II, SA and TPO are given in Figures 4.5, 4.6 and 4.7 respectively. These parameters were chosen after extensive experimentation and by following guidelines given in the literature [1, 59, 57, 22]. TPO needed a lot of tuning effort compared to the other two algorithms. The performance of TPO depends on the performance of both the II and SA phases and hence more effort is needed to balance the two phases. Also, it was noticed that the performance of TPO is very sensitive to the initial temperature of the SA phase, in addition to the number of local optimizations of the II phase. For deciding the local minimum in II, the same approximation was used as by Ioannidis [22]. A state is considered to be an r-local minimum if the cost of that state is less than that of the cost of n randomly chosen neighbors (with repetition) of that state. Here, n was chosen to be the number of edges in the condition graph of a rule. This is equivalent to the maximum number of β nodes in any Gator network for that rule and hence is an upper bound on the number of times a create-beta or a kill-beta can be applied. Deciding a local minimum by exhaustively searching the neighbors of a state is an expensive process and hence we believe that the choice of using an r-local minima is a more practical one.

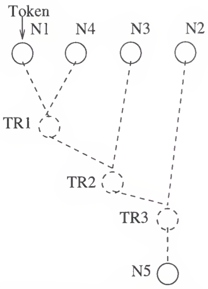
The Gator network data structures and the three randomized search algorithms (II, SA and TPO) were implemented in Ariel using the E-language, which is a persistent version of C++ [41, 40]. The optimizer creates a complete new Gator network every time it applies a local change operator. The Gator network nodes are made persistent by using "dbclasses" [41] provided by E. Please refer to [31] for more implementation details on the Gator network optimizer module.

4.6 Generating Token Join Order Plans

Every node in a Gator network (except the P-node) has a join plan attached to it. The join plan specifies the order in which a token arriving at the node would be joined with each of its siblings. For example, in Figure 4.8(a) the join plan attached to node N1 is (N4,N3,N2). When a token arrives at node N1, it is first joined with the contents of node N4. The resulting Temporary Result (TR) of the join is then joined with contents of the node N3 and so on, as shown in Figure 4.8(b). The temporary results are not materialized. They are generated during the token propagation process and discarded after that. It is important to choose a join plan with the minimum cost. However, it is too expensive to use the traditional query optimization techniques to find the join order plan because the join order plan must be chosen for all the nodes in each of the generated Gator networks during the Gator network optimization process. Instead, the following heuristic is used: during each



(a) Gator network with join order plan for N1=(N4,N3,N2).



(b) Propagating a token arriving at node N1.

Figure 4.8. Join order plan for a Gator node

of the two-way joins, the current result should be joined with the smallest connected sibling. This gives a reasonable join order plan quickly.

4.7 Optimizer Characteristics and Performance

This section presents the details of various experiments conducted to study the relative behavior of II, SA and TPO for various rules under different update frequency distributions, catalogs and cost models. Rules were created on synthetically generated databases. Three different databases were generated. The properties of catalogs used to generate the databases are given below.

	Relation Cardinality	% of unique values in attributes
Catalog 1	[1000, 100000]	[90, 100]
Catalog 2	[10, 100] - 20%	(0, 20) - 70 %
	[100, 1000] - 64%	[20, 100) - 5%
	[1000, 10000] - 16%	100 - 25%
Catalog 3	[1000, 10000]	[90, 100]

The table gives the cardinality distribution for tables in each catalog. Every table has a primary key attribute. For the other attributes, the table shows the percentage of attributes which fall in each cardinality range. E.g. for Catalog2, 64% of the tables have between 100 and 1000 tuples. Furthermore, 5% of the attributes have at least 20% and fewer than 100% as many unique values as the primary key.

Indices were created only on large relations in the first database. Experiments were performed on rules having the following types of Rule Condition Graphs (RCGs):

Chain type Each relation in the rule condition participates in a join with two other relations such that the rule condition graph looks like a string. The two relations at the two ends of the string participate in only one join.

The following is an example of a rule with a string type RCG. R1, R2, R3, R4 and R5 are relations used only for purposes of illustrating the different types of rule graphs.

```
define rule Rule1

if R1.a = R2.b and R2.c = R3.d and R3.d = R4.e

then <action1>
```

Star type One relation participates in a join with all the other relations in the rule condition. The following is an example of a rule with a star type RCG.

```
define rule Rule2
if R1.a = R2.b and R1.c = R3.c and R1.b = R4.d
then <action2>
```

Random type Joins between relations are chosen randomly to create a connected rule condition graph with no cycles. Here is an example of a rule with a random type RCG:

```
define rule Rule3
if R1.a = R2.b and R1.c = R3.c and R1.b = R4.d and R4.d = R5.e
then <action3>
```

The update frequency distribution of various relations in the database significantly affects the performance of discrimination networks. The following three update frequency distributions were chosen:

Skewed One of the relations has a very high update frequency and the other relations have low frequencies.

Even All the relations have the same update frequency.

Step The update frequencies of relations decrease in a stair-like manner.

The actual frequencies used are summarized in the table below. In all cases, frequencies sum to one.

	Equal	Step	Skew
5 relations	0.2 each	0.4, 0.3, 0.2, 0.05, 0.05	0.8, 0.05 others
10 relations	0.1 each	0.4, 0.3, 4 each with 0.05	0.7, 0.124, 0.022 others
		and 0.025	
15 relations	0.067 each	0.3, 0.2, 0.14, 0.04, 0.03,	0.6, 0.14, 0.02 others
		and 4 each with 0.02	

The *size* of a rule is the number of tuple variables in its condition. Rules of size 5, 10 and 15 were created with string, star and random type rule condition graphs. Each one of these rules was tested with equal, step and skewed frequencies, with cost models CM1 and CM2 and catalogs Catalog1 and Catalog2. For a number of (RCG, size, frequency, cost model, catalog) combinations, each of TPO, SA and II was run 10 times with different random seeds. For each algorithm, the average of the output state in all the 10 runs was computed. II was run the same amount of time as was taken by TPO. For each of these cases and for each algorithm, the average scaled cost was computed by dividing the average cost of 10 runs of that algorithm by the best state found by all the runs of all the algorithms for that case.

The results of various experiments conducted are shown in Figures 4.10 through 4.18. It can be seen that for rules with size 5, irrespective of the catalogs, cost models and frequencies, all the algorithms are doing well. There is no difference in the costs of states produced by different algorithms. As the rule size increases from 5 to 15, the difference in the relative behavior of the algorithms also increases. Also, as the rule size increases, the average behavior of the algorithms tends to go away from the ideal value of 1, that is the algorithms become less stable. In general, TPO performs better than II and SA and there is no clear winner between II and SA. Both II and SA are the worst of the three in some cases. No significant difference was observed in the behavior of the algorithms for the cost models CM1 and CM2. It was observed that when considering the best of all runs in the experiments, all of II, SA and TPO performed well.

It can be noticed that even though TPO performs better than II and SA in a majority of cases, there are instances where II and SA perform better than TPO (e.g. 4.11(C), 4.18(B) 4.10(A) and 4.16(C)). These graphs also illustrate the difficulty in tuning TPO to perform well in all cases. It can also be noticed that wherever II or SA is doing better than TPO, there is another algorithm (SA or II, respectively) performing worse than TPO. In the following discussion, Graph 4.18(B) is chosen as the representative of all the graphs where II is doing better than TPO and Graph 4.10(A) is chosen to represent the graphs where SA is doing better than TPO. The behavior of the algorithms in graphs 4.18(B) and 4.10(A) is explained next and some general conclusions about the overall behavior of the algorithms are given after that.

In graph 4.18(B), II is giving better result quality for fifteen tuple variable rules than TPO. Here, the problem is in deciding the crossover point between II and SA in TPO. This decision is crucial, especially when the search space contains many local minima at high-cost states with a small but significant portion of them at low-cost states (space A2, similar to the search space of left deep query trees in [23]). In this case, doing a few iterations in the II phase of TPO might leave the starting state of SA at a high-cost state (because there are many local minima at high-cost states) making the overall result of TPO not satisfactory. Doing more iterations or local optimizations in II is always beneficial in this case, because that helps to find a low-cost local minimum. The presence of many high-cost local minima also explains the behavior of SA in this case. SA searches high-cost states when the temperature is high, and when the temperature gets low, it reaches a local minimum state and searches around that state. Since there are many local minima at high-cost states, SA also can get trapped in one of the high-cost states and hence its performance is not good. In this case, if the number of iterations in the II phase of TPO is increased, then TPO is going to do at least as well as II.

In graph 4.10(A), SA is optimizing more effectively than TPO and II is doing worse than TPO. Here, partly, the problem is in estimating the initial temperature of the SA phase of TPO. Here, many runs of II generate a high-cost local minimum and hence its performance is not good. TPO seems to extricate itself out of these high cost states in many of the cases but not all. The reason seems to be that the cost of states separating the low-cost states is not very low and hence the initial temperature ($0.5^{*}best_{II}$, here $best_{II} < 20000$) seems to be not enough to jump over those states. Also, the low value of SA shows the presence of low cost local minima. In fact, for this case (graph 4.10(A)) when we repeated the experiments with high initial temperature ($1.0^{*}best_{II}$) the average value of TPO came down to 1.307 compared to the current value 1.405 and the SA value of 1.230. In general, we noticed that the behavior of TPO is very sensitive to the initial temperature and we tuned it carefully for various cases. Here, part of the reason that TPO is not doing well could also be due to the existence of many local minima at high cost states. This is because the average behavior of SA is also not close to 1 which suggests the possibility of many high-cost local minima.

In general, it can be stated that TPO does well and the performance of II and SA are close to TPO (within 10-20% in most cases). A possible explanation for this is that the search space, in general, contains most of the local minima at low-cost states with reasonably high cost states separating the local minima.

Each iteration of II generates a random state and follows downhill moves until it reaches a local minimum. Since most of the local minima are at low-cost states, II is able to find a good local minimum. SA explores high-cost states when the temperature is high and reaches the local minima states at low temperatures and searches around those states. Since, again, there are many local minima at low states, it is able to find a good one in most of the cases. TPO starts with a good local minimum state and performs search starting with low initial temperature. It seems that, since the temperature is low, it is able to search or come across a lot of low-cost states and hence it has a high chance of finding a better state. Also, in other experiments, it was noticed that the SA phase of TPO does not do well when the starting temperature is very low. Based on this, coupled with graph 4.10(A), it can be said that the cost of states separating the low-cost states in reasonably high (for states with cost < 20000, the SA phase of SA never performs well when the starting temperature is less than $(0.5{*}best_{II})$).

Graphs 4.14(B), 4.15(B), 4.18(A), and 4.12(B) illustrate cases where the performance of SA is worse compared to II and TPO. II is doing well in these situations, which means that there are enough valleys containing a low cost minimum so that II can almost always find a good overall solution. SA seems to be getting trapped in high-cost states and does not reach the low-cost local minimum states. The solution space in these cases seem to contain high-cost valleys and the temperature does not seem to be high enough to escape these valleys.

Even though II and SA are performing close to TPO in most of the cases, the ability of TPO to avoid worst-case behavior makes it the winner. It is used in the next chapter for generating the optimal Gator network to compare with optimal Rete and TREAT.

Tables showing the average optimization time in seconds taken by TPO and SA for all the cases are shown in Figure 4.9. The time taken by II is not shown here because it was given the same amount of time as TPO. Except in a few cases, TPO takes less time than SA. The difference in the time taken by TPO and SA increases as rule size rises from 5 to 10. At size 15, both II and SA take almost the same time. Again, no significant differences were found between the optimization times of the two cost models.

4.8 Limitations of the Current Implementation of the Ariel Gator Network Optimizer

The current implementation of the Ariel Gator network optimizer is not efficient because of the following reasons:

1. In the Gator network optimizer, the neighbor of a Gator network is generated by making a duplicate of the current one. An alternative and more efficient way would be to directly modify the current network to the next state and to undo the changes when the new state is not accepted.

2. The Gator network nodes are implemented by using the dbclasses [41, 4] of the E-language (instead of regular C++ classes). This was done to provide persistency to the optimized Gator network nodes. This slowed down the optimizer performance because dereferencing a dbclass pointer is more expensive than a main memory pointer. Also, the E-compiler used to compile the code was not of commercial quality.

Because of these reasons, we believe that the Gator network optimizer can be speeded up by a factor of 10 or more without changing the algorithms used for search. [27] presents initial results which support this statement.

4.9 Conclusion

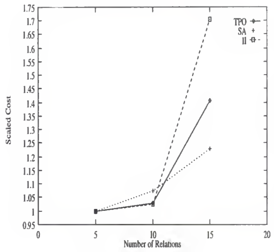
This chapter proposed a randomized algorithms-based strategy for optimizing Gator networks. Randomized algorithms were chosen to deal with the problem of large search space (the search space of a Gator network optimizer is much higher than that of a query optimizer). Three randomized algorithms, II, SA and TPO, were applied for the Gator network optimization problem. Experimental results showed that TPO performs better, in terms of output quality, compared to the other algorithms for the Gator network optimization problem. The output quality of the algorithms was close, but TPO was best or second best in every test. TPO required a lot of

	star,step,CM1, Catalog 1			star,step,CM1, Catalog 2			star,step,CM2, Catalog 1		
Size	5	10	15	5	10	15	5	10	15
TPO	39	215	598	29	196	600	36	230	600
SA	47	247	600	28	201	537	44	243	600
	star,step,CM2, Catalog 2			string,eq,CM1, Catalog 1			random,skew,CM2, Catalog 2		
Size	5	10	15	5	10	15	5	10	15
TPO	40	237	600	33	191	530	39	209	583
SA	37	271	600	40	219	556	48	204	587

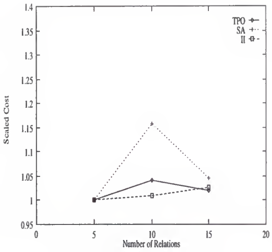
Figure 4.9. Average optimization time (in seconds) of TPO and SA

tuning effort compared to the other two algorithms and its performance was very sensitive to the initial temperature of the SA phase, in addition to the number of local optimizations of the II phase. Based on the relative behaviour of the three algorithms, it was conjectured that the search space, in general, contains most of the local minima at low cost states with reasonably high cost states separating the local minima.

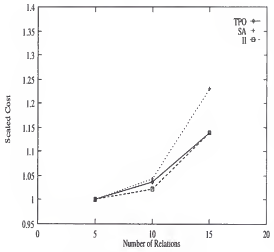
This concludes the discussion of the Gator network optimizer. Chapter 5 compares the performance of Gator with that of Rete and TREAT and it also validates the Gator network cost model.



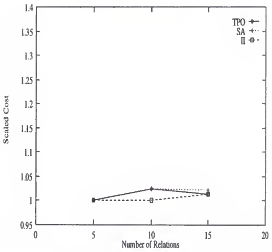
(A) CM1, Catalog 1



(B) CM1, Catalog 2

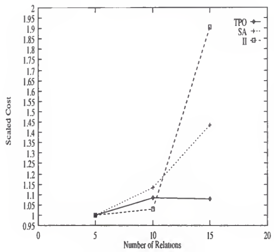


(C) CM2, Catalog 1

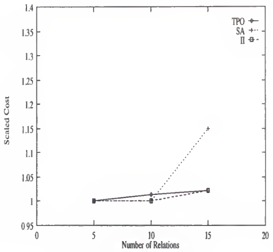


(D) CM2, Catalog 2

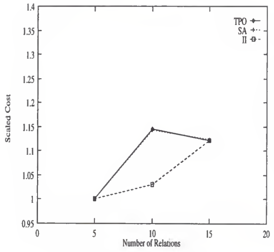
Figure 4.10. Chain type RCG and equal frequency distribution



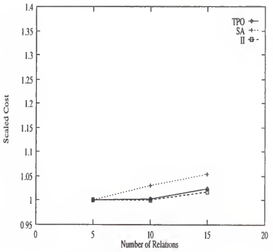
(A) CM1, Catalog 1



(B) CM1, Catalog 2

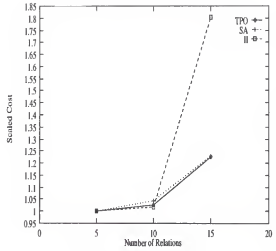


(C) CM2, Catalog 1

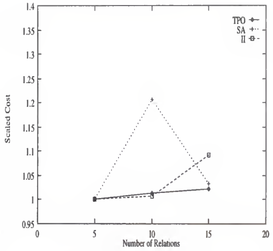


(D) CM2, Catalog 2

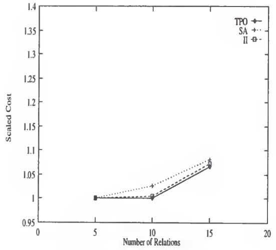
Figure 4.11. Chain type RCG and step frequency distribution



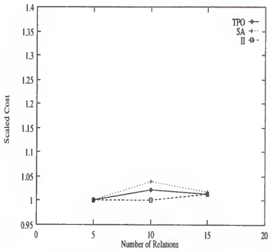
(A) CM1, Catalog 1



(B) CM1, Catalog 2

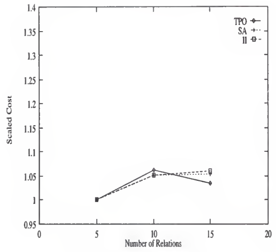


(C) CM2, Catalog 1

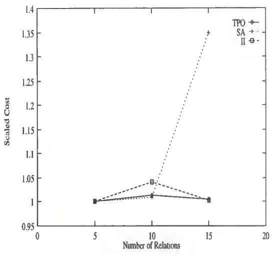


(D) CM2, Catalog 2

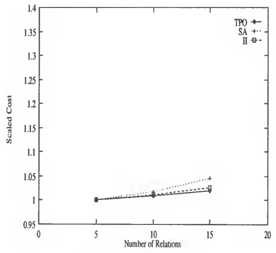
Figure 4.12. Chain type RCG and skew frequency distribution



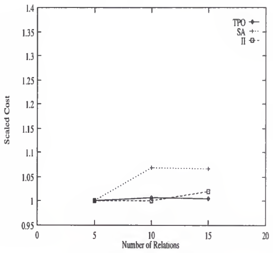
(A) CM1, Catalog 1



(B) CM1, Catalog 2

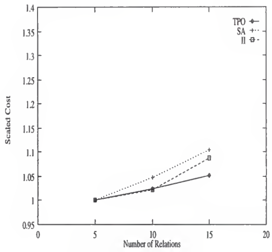


(C) CM2, Catalog 1

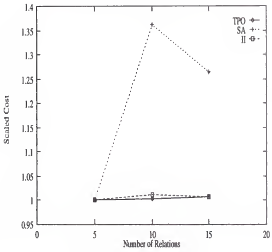


(D) CM2, Catalog 2

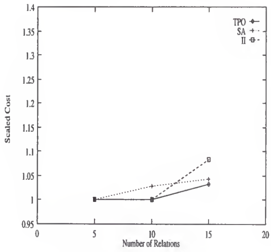
Figure 4.13. Star type RCG and equal frequency distribution



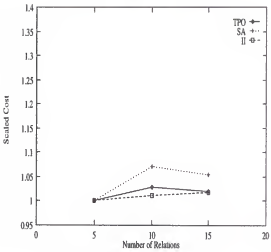
(A) CM1, Catalog 1



(B) CM1, Catalog 2

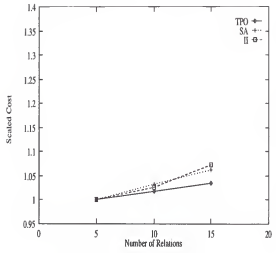


(C) CM2, Catalog 1

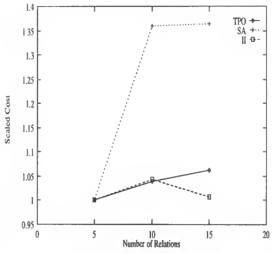


(D) CM2, Catalog 2

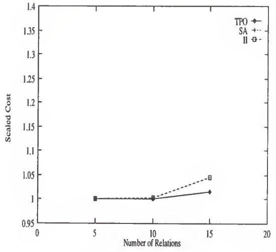
Figure 4.14. Star type RCG and step frequency distribution



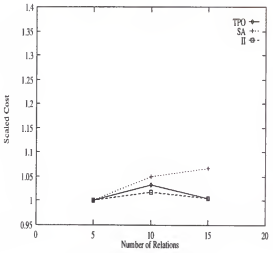
(A) CM1, Catalog 1



(B) CM1, Catalog 2

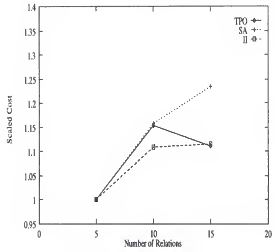


(C) CM2, Catalog 1

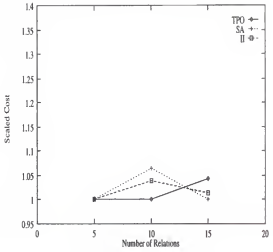


(D) CM2, Catalog 2

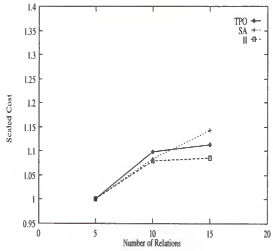
Figure 4.15. Star type RCG and skew frequency distribution



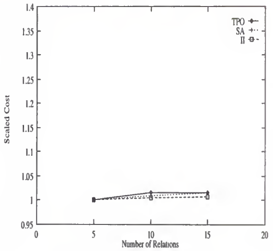
(A) CM1, Catalog 1



(B) CM1, Catalog 2

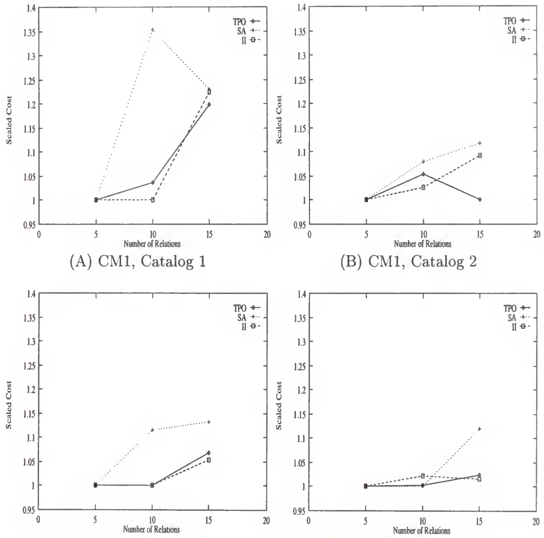


(C) CM2, Catalog 1



(D) CM2, Catalog 2

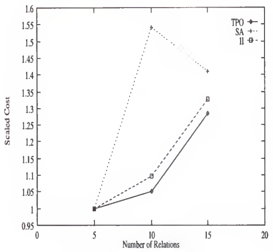
Figure 4.16. Rand type RCG and equal frequency distribution



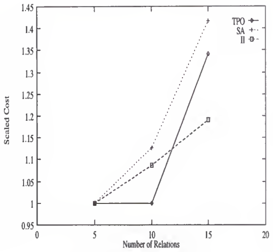
(A) CM1, Catalog 1 (B) CM1, Catalog 2

(C) CM2, Catalog 1 (D) CM2, Catalog 2

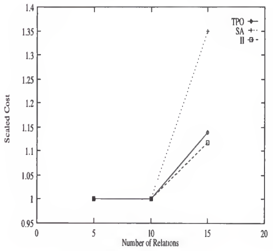
Figure 4.17. Rand type RCG and step frequency distribution



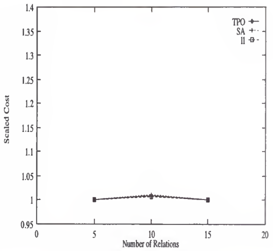
(A) CM1, Catalog 1



(B) CM1, Catalog 2



(C) CM2, Catalog 1



(D) CM2, Catalog 2

Figure 4.18. Rand type RCG and skew frequency distribution

CHAPTER 5
VALIDATION OF GATOR NETWORK COST FUNCTIONS

The Gator network cost model presented in Chapter 3 estimates the cost of a Gator network relative to other Gator networks for a particular trigger. The optimization strategy presented in Chapter 4 uses the Gator network cost model to compare different Gator networks while performing search for an optimized Gator network for a given rule. The cost functions and the optimization strategy have been implemented as part of the Ariel active relational DBMS. Various experiments were conducted on Ariel with two motives: 1. compare the relative performance of Gator, Rete, and TREAT networks, and 2. validate the cost model against the actual pattern matching time of Gator networks. This chapter presents a detailed report on these experiments. Additional discussion on the results of these experiments can be found in [7].

The performance metric in all the experiments is the *rule condition evaluation time* or the *rule activation time.* The rule condition evaluation time is the time to evaluate a rule condition using a discrimination network. The average rule activation time was measured by processing a randomly generated stream of updates. The table to which an update was applied was determined using a frequency distribution equivalent to the update frequency statistics maintained in the system catalog. Inserts were performed on each table, and the token testing time for each was measured. The "total rule condition testing time" was calculated by multiplying the time spent propagating a token for each table by the update frequency of that table.

Rules were created on synthetically generated databases. The details of the catalogs that were used to generate the databases are given in Chapter 4. Rules were created based on the following types of rule condition graphs: String, Star and Random. For each type of rule condition graph, rules of size 5, 10 and 15 were created. For each rule size, a number of rules were created based on the placement and the number of selection conditions in the rules. Each of these rules were tested with Equal, Skewed and Step update frequency distributions. As the relation sizes are small in Catalogs 2 and 3, no indexes were created. B^+-tree indexes were created on primary keys of large relations in Catalog 1.

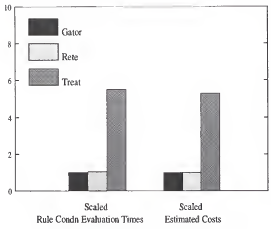
In all the experiments, an optimized Gator network was compared with a TREAT network and an optimized, left-deep Rete network. For each rule, an optimized Gator network was generated by using the Two-Phase optimization strategy. Rete networks were optimized by using a dynamic programming-style Rete network optimizer. Optimized Rete networks were compared with the optimized Gator networks in order to give a fair comparison between Gator and Rete. Due to the fact that Exodus does not allow buffer space to be set very low, it was not possible to test token propagation cost for cost model 2. All the experiments were run on a Sun SPARCstation 5/110.

5.1 Comparison Between Gator, Rete and TREAT

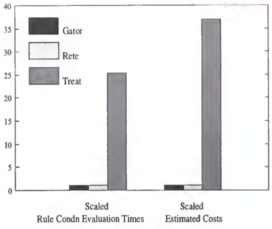
Figures 5.1, 5.2 and 5.3 show relative estimated vs. actually measured token propagation times for Gator, Rete and TREAT networks for rules having 5, 10 and 15 tuple variables respectively. Each figure shows a sample of the results of experiments conducted for various (catalog,RCG,frequency) combinations on rules of the same size.

There is no significant difference in the general behavior of the three networks among rules of different sizes. In general, the performance of Gator is significantly better than that of TREAT and reasonably better than that of Rete. TREAT has the highest average token propagation time of the three networks. In some cases, its token propagation time is higher than that of Gator by a few orders of magnitude. The performance of the optimized Rete is in between that of Gator and TREAT. In all the three figures, there are cases where the performance of the optimized Rete is close to that of the optimized Gator.

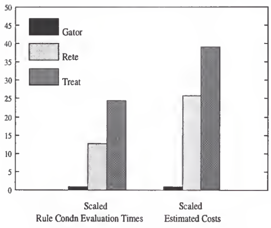
TREAT requires evaluating all the join operations in the rule condition for every update and hence its cost is higher. Gator and Rete avoid some of the join operations by utilizing the pre-computed results stored in the beta nodes. Since the cost model (CM1) assumes the availability of large buffer space, the maintenance cost of beta nodes in Rete and Gator (due to the incoming tokens) is low. The maintenance cost is low because updates to nodes (either alpha or beta) do not result in their pages to be written to disk at transaction commit time (because of large buffer space). Only log pages are written to disk at commit time and the node updates are assumed to be



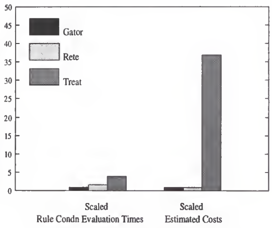
(A) String type RCG with eq frequency distribution



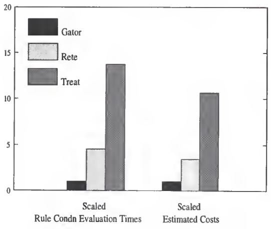
(B) Star type RCG with skew frequency distribution



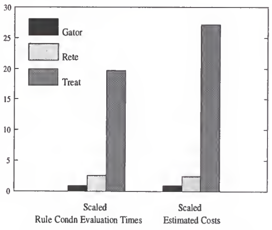
(C) String type RCG with step frequency distribution



(D) Star type RCG with skew frequency distribution

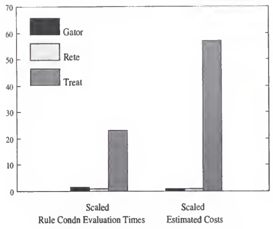


(E) String type RCG with step frequency distribution

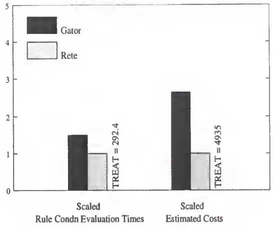


(F) Random type RCG with skew frequency distribution

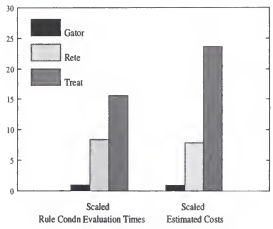
Figure 5.1. Comparison of estimated costs and condition evaluation times for rules of size five



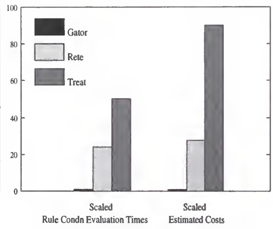
(A) Star type RCG with step frequency distribution



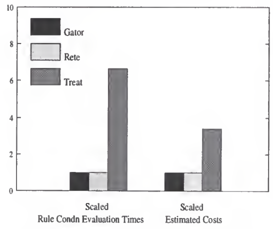
(B) Random type RCG with skew frequency distribution



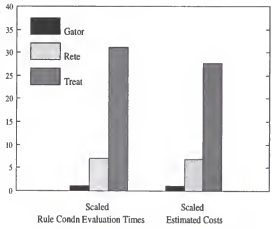
(C) String type RCG with eq frequency distribution



(D) Random type RCG with step frequency distribution

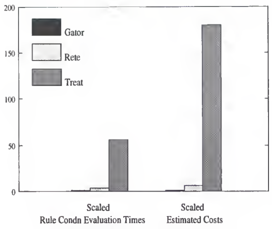


(E) Star type RCG with eq frequency distribution

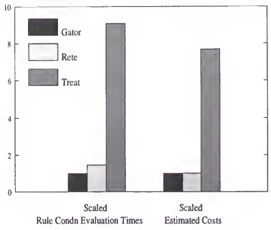


(F) Random type RCG with step frequency distribution

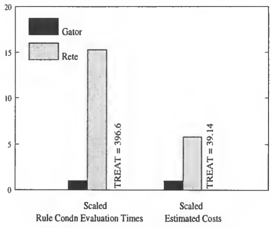
Figure 5.2. Comparison of estimated costs and condition evaluation times for rules of size ten



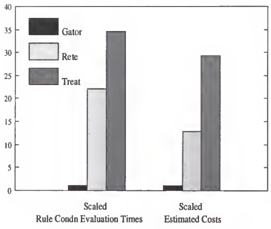
(A) String type RCG with eq frequency distribution



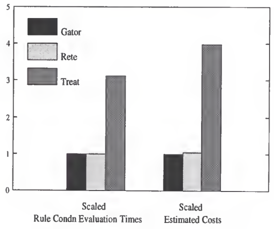
(B) Star type RCG with step frequency distribution



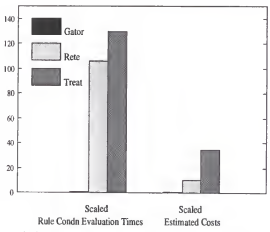
(C) Random type RCG with step frequency distribution



(D) String type RCG with eq frequency distribution



(E) Star type RCG with step frequency distribution



(F) Random type RCG with step frequency distribution

Figure 5.3. Comparison of estimated costs and condition evaluation times for rules of size fifteen

transmitted to disk at the checkpointing time. The above factors explain the overall superiority of Gator and Rete over TREAT in the experiments. In case of Gator, the optimizer has the flexibility in choosing the number of beta nodes depending on the update frequency and other database statistics. Only those beta nodes that are useful or that are going to help in lowering the overall cost are created in a Gator network. The experiments also reveal that the optimized Gator network shape in most of the experiments is neither pure Rete nor TREAT, but an intermediate form having a few beta nodes (less than the number of beta nodes in a Rete network).

Figures 5.1(B) and 5.2(E) show interesting results where the optimized Gator network generated by the Gator network optimizer is the same as the optimized Rete network. This shows the flexibility of Gator networks and also, the efficacy of the Gator network optimizer. When Rete or TREAT network is an optimal one for a given rule, Gator takes that network shape.

It can also be noticed that in all the experiments (except 5.2(A) and (B), which are discussed later), the cost of the optimized Gator network generated by the Gator network optimizer is less than that of Rete and TREAT networks. This shows that there exists a Gator network which can perform better than (the optimal) Rete and TREAT for rules under any of the parameter values. This work also demonstrates the feasibility of finding such a Gator network in a reasonable amount of time by using randomized optimization algorithms.

It is interesting to compare the best and worst case performance of Rete and TREAT with respect to Gator along the different dimensions of interest: rule size, rule condition graph, catalog and update frequency distribution. The best-case performance of a network (Rete or TREAT) along a dimension gives the lowest average token propagation time of that network relative to that of Gator, from among the experiments conducted on rules having a fixed value for that dimension and different values for the other dimensions. For example, the best case performance of TREAT for "rule size = 5" in Figure (5.5) is 1.38. It is the lowest relative token propagation time of TREAT from among all the experiments conducted on rules of size 5 with different RCGs, different catalogs and different update frequency distributions. The worst-case performance of a network (Rete or TREAT) along a dimension gives the highest average token propagation time of that network relative to that of Gator, from among the experiments conducted on rules having a fixed value for that dimension and different values for the other dimensions.

The best-case performance by Rete and TREAT along the different dimensions is given in the tables in Figures 5.4 and 5.5 respectively. It can be observed that their performance is close to that of Gator for all values in each dimension. That means, there exist Rete and TREAT networks for some rules in each dimension whose performance is comparable to that of Gator. For some cases, the token propagation time of Rete is less than that of Gator. For these cases, the estimated cost of Rete is higher than that of Gator and the two values are very close. When the estimated costs of the two networks are so close, errors of this sort can be expected. It was observed that whenever the difference between the estimated costs of networks (Gator, Rete or TREAT) was reasonably high, their rule condition evaluation times followed the same order as that of their respective estimated costs.

The worst-case performance by Rete and TREAT with respect to Gator for different rule sizes and rule condition graphs is given in tables 5.6 and 5.7 respectively. It can be observed that the worst-case behavior increases with rule size for both Rete and TREAT. Also, the condition testing times for TREAT are much higher than those of Rete.

The worst-case performance of Rete and TREAT with respect to Gator is higher for rules having string and Random type RCGs than those having the star RCG. The performance of the optimized Rete is close to that of the optimized Gator for rules having the star RCG (the maximum ratio noted is 3.96). Since Gator networks do not create beta nodes that involve a cartesian product, the Gator network shape for star type RCG rules tends to be left-deep and hence the Gator network performance in this case is close to that of Rete.

The different update frequency distribution values and catalog statistics do not seem to have a significant effect on the worst-case performance of Rete and TREAT. The worst-case performance of Rete and TREAT is the same for different values along the update frequency and the catalog dimensions and are not shown here.

Figures 5.2(A) and 5.2(B) show cases where the cost of the optimized Gator is higher than that of the optimized Rete. Here, the randomized optimizer was not able to come up with the optimal Rete for the optimized Gator. Occasional errors like this are to be expected in randomized algorithms. However, it was observed during

Rule Size	**5**	**10**	**15**
	0.6	0.19	0.32
RCG	**String**	**Star**	**Random**
	0.83	0.32	0.66
Catalog	**Catalog1**	**Catalog2**	**Catalog3**
	0.19	0.32	0.6
Update Frequency Distribution	**Eq**	**Skew**	**Step**
	0.19	0.66	0.66

Figure 5.4. Best-case performance by Rete along different dimensions

Rule Size	**5**	**10**	**15**
	1.38	1.02	1.91
RCG	**String**	**Star**	**Random**
	3.78	1.02	1.38
Catalog	**Catalog1**	**Catalog2**	**Catalog3**
	1.38	1.02	1.42
Update Frequency Distribution	**Eq**	**Skew**	**Step**
	1.02	1.38	1.44

Figure 5.5. Best-case performance by TREAT along different dimensions

the various test that were run that it was a rare occurrence. One point in favor of the Gator network optimizer in these cases is that the Gator network optimizer took about half the time which the Rete network optimizer took to arrive at the optimal Rete network. The search space for star type RCGs is high [37] and hence the Rete optimizer based on dynamic programming takes a lot of time to come up with the optimal Rete network. Also, even in these cases, the relative order among the rule activation times of Gator, Rete and TREAT is the same as the one given by the cost functions.

Rule Size	5	10	15
	25.17	48.17	105.74
RCG	**String**	**Star**	**Random**
	48.17	3.96	105.74

Figure 5.6. Worst-case performance by Rete along different dimensions

Rule Size	5	10	15
	308.48	513.83	1821.9
RCG	**String**	**Star**	**Random**
	308.4	46.7	1821.9

Figure 5.7. Worst-case performance by TREAT along different dimensions

5.2 Accuracy of the Gator Network Cost Model

From Figures 5.1, 5.2 and 5.3, it can be said that the cost functions estimated the costs of various discrimination networks reasonably well. Even though the (ratio of) estimated costs and the (ratio of) actual token propagation times do not match in some cases, the relative order among the estimated costs and the token propagation times of Gator, Rete and TREAT is always the same (except when the estimated costs are very close, as in Table 5.4).

It can be observed that the estimated costs and the actual rule condition evaluation times are not proportional for TREAT in some cases (e.g. 5.1(D), 5.2(B) and 5.3(A)). In these cases, the scaled cost of the TREAT network is overestimated. Since the scaled cost of TREAT is the ratio between the estimated costs of TREAT and Gator, the problem could be because of either or both of the following: 1. overestimation of TREAT network cost. 2. underestimation of Gator network cost.

Further analysis revealed that this is due to the overestimation of the TREAT network cost. Estimating the cost of a TREAT network involves estimating the cost of performing a sequence of join operations due to tokens fed into its leaf nodes. Intermediate relations are created for the temporary results formed during the join process and are used to join with the other nodes of the TREAT network. The cost model has overestimated the size of temporary results which resulted in overestimating the cost of a TREAT network. The cost model assumes the availability of large buffer space and the processing cost of a multi-way join in this cost model is very sensitive to the size of the intermediate join results. Accurate estimation of temporary result sizes is challenging because of the way errors in join selectivity estimation propagate during a long join sequence [21].

Figures 5.8 and 5.9 show the estimated temporary result sizes and the actual temporary result sizes in a TREAT network for rules having Random and Star type of Rule Condition Graphs. The Random RCG rule had 10 tuple variables in its rule condition and was tested with Step frequency distribution. The Star RCG rule also had 10 tuple variables in its rule condition and was tested with Skew frequency distribution. The temporary result sizes are shown only for those join sequences that resulted due to tokens entering at virtual alpha nodes in those rules. The overall join processing cost due to tokens entering at stored alpha nodes in those rules is negligible and hence the temporary result sizes in those join sequences are not shown here. In Figure 5.8, the difference between the estimated costs and the token propagation times is small and it is reflected in the temporary result sizes also. In Figure 5.9, the relative estimated costs and the relative token propagation times are way-off. Here, it can be seen how the temporary results are overestimated for TREAT.

The difference between the estimated costs and the actual measured times of TREAT is higher in Catalogs 1 and 2 than in Catalog 3. This seems to be due to higher relation sizes and lower fraction of unique values of join attributes in relations in Catalogs 1 and 2.

However, in case of Gator networks and Rete networks the difference between the estimated sizes of temporary results and the actual sizes of temporary results is not as high as in TREAT networks. This seems to be because of the way the beta nodes are created in these networks. In Gator and Rete networks, stored alpha nodes are clubbed with virtual nodes to form beta nodes. This results in beta nodes having smaller size than the size of their parent virtual alpha child nodes. Since the size of the beta nodes is small, the error in estimating the join result size tends to be low.

When the estimated sizes of beta nodes are high, more deviation in the sizes of estimated and the actual temporary results can be expected in Gator and Rete networks also. The join selectivity factor affects not only the join processing cost but also the estimated sizes of beta nodes and the insert and delete frequencies of beta nodes and hence has a significant effect on the accuracy of the estimated cost of a discrimination network. The join result size estimation (which includes the estimation of the number of unique values of join attributes in temporary results) in the Gator network cost model is based on the assumptions of uniform distribution of values and independence of value distribution in different columns of relations and is primitive. Better techniques to estimate the selectivities will help in improving the accuracy of the estimated costs of discrimination networks.

5.3 Explanation of the Optimized Gator Network Shape

The optimized Gator and Rete networks generated for the rule condition graphs in Figures 5.10(A) and 5.11(A) are shown in Figures 5.10(B) and 5.11(B) respectively. Both the rules were tested with Step frequency distribution. In all the networks, virtual alpha nodes are created for relations with no selection predicate in the rule condition, preventing the duplication of relations and thus saving space. In the case of the Gator network in 5.10(B), the alpha nodes with high update frequency are pushed down the discrimination network, towards the root node or the P-node. This means that fewer token joins need to be performed as tokens propagate through the network due to updates. It can also be noticed that the stored alpha nodes with low size are at the top of the network. This helps to reduce the size of the β nodes below them. In the case of the Gator network in 5.11(B), the relation D which has the highest update frequency (0.4) is closer to the P-node. But, the relation H (update frequency=0.3) is at the top of the network. This is to reduce the size of beta nodes. The relation H has a stored alpha created for it and is joined with G to reduce the size of the beta node below it. Another observation is that, in both the Gator networks, the optimizer has not created a beta node with two virtual alpha nodes as child nodes, since that could potentially form a beta node with large size.

In the case of the Rete network in Figure 5.10(B), it can be noticed that the alpha node corresponding to the relation D is at the top of the network. Since the

```
                                 Gator  :  TREAT
   Ratio of estimated costs:       1    :   90
   Ratio of rule condition
   evaluation times:               1    :   50


No.of matches in the join sequence due to
a token at relation B:

Estimated -  1, 1, 1, 1, 1, 1, 1, 1, 1
Actual    -  1, 1, 1, 1, 2, 4, 2, 3, 6

No.of matches in the join sequence due to
a token at relation D:

Estimated -  1, 1, 1, 1, 1, 1, 1, 1, 1
Actual    -  1, 1, 2, 2, 1, 2, 4, 2, 2

No.of matches in the join sequence due to
a token at relation G:

Estimated -  3, 1, 1, 15, 15, 144, 1, 1, 1
Actual    -  1, 1, 2, 2,  32, 2,   2, 2, 4

No.of matches in the join sequence due to
a token at relation E:

Estimated -  1, 1, 1, 1, 1, 9, 1, 1, 1
Actual    -  1, 4, 4, 4, 4, 8, 4, 4, 8

No.of matches in the join sequence due to
a token at relation I:

Estimated -  1, 1, 15, 15, 15, 144, 1, 1, 1
Actual    -  1, 1, 4,  4,  1,  1,   2, 1, 2
```

Figure 5.8. Estimated and actual temporary result sizes during token propagation in a TREAT network for a rule with 10 tuple variables, random RCG and step frequency distribution

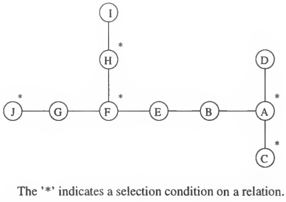
```
                                   Gator  :   TREAT
  Ratio of estimated costs:          1    :    2131
  Ratio of rule condition
  evaluation times:                  1    :    4.67


No.of matches due to a token at relation D:

Estimated -  1, 1, 1, 1, 1, 15, 144, 144, 1382
Actual    -  1, 1, 2, 2, 2, 2,  2,   2,   4


No.of matches due to a token at relation E:

Estimated -  1,  1, 1, 1, 1, 9, 9, 430, 3070
Actual    -  10, 1, 2, 2, 2, 2, 2, 4,   4


No.of matches due to a token at relation G:

Estimated -  1, 1, 1, 1, 1, 15, 144, 144, 1028
Actual    -  10, 1, 2, 2, 2, 2, 2, 4, 4


No.of matches due to a token at relation I:

Estimated -  1, 1, 1, 1, 1, 15, 144, 1382, 9868
Actual    -  1, 1, 2, 2, 2, 2,  2,   2,    2


No.of matches due to a token at relation B:

Estimated -  1, 1, 1, 1, 1, 15, 15, 717, 5119
Actual    -  1, 1, 2, 2, 2, 2,  2,  4,   4
```

Figure 5.9. Estimated and actual temporary result sizes during token propagation in a TREAT network for a rule with 10 tuple variables, star RCG and skew frequency distribution

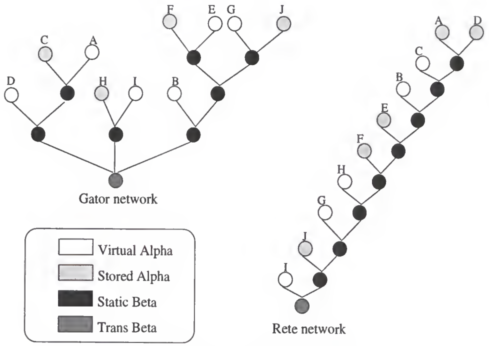
relation D has a high update frequency, we would expect it to be near the bottom of the network. However, when it was forced to be near the bottom, the cost of the generated network was higher than that of the one given by the optimizer. Further, the token propagation times of the two Rete networks was tested by passing tokens through them. The Rete network generated by the optimizer took less time than the Rete network obtained by forcing D to be near the bottom (the network which we intuitively expect to perform better). This illustrates the importance of using a cost-based optimizer, rather than just using heuristics to find a better network.

Figure 5.12(B) shows the optimized Gator and the optimal Rete networks generated for the Star RCG in Figure 5.12(A) with Equal frequency. In the case of the Gator network, it can be noticed that each beta node has a maximum of one beta node as its child node. This is because Gator networks do not allow the creation of a beta node that requires a cross product among its child nodes. The relative order of the relations in the Rete network in 5.12(B) is almost the same as the ones in the Gator network and their costs are also close (cost of Gator = 821 and cost of Rete = 823).

Figure 5.13 shows the Gator networks generated for a rule with string RCG, 15 tuple variables and Equal frequency distribution for the two cost models. The figure also shows the optimal Rete and TREAT networks for that rule. The Gator network of the cost model 2 has fewer beta nodes. Also, the beta node has a higher fan-out resulting in a shorter leaf-to-root distance. In cost model 2, there is a limited buffer space and the cost of maintaining the beta nodes is significant. Hence, the

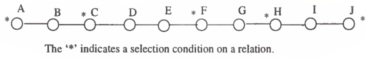


(A) Random type RCG with ten relations.

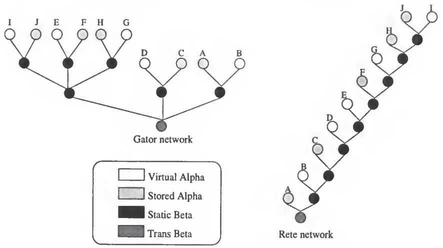


(B) Gator and Rete networks.

Figure 5.10. A random type RCG of size 10 and its optimized Gator and Rete networks

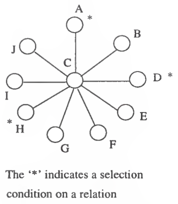


(A) String type RCG with ten relations.

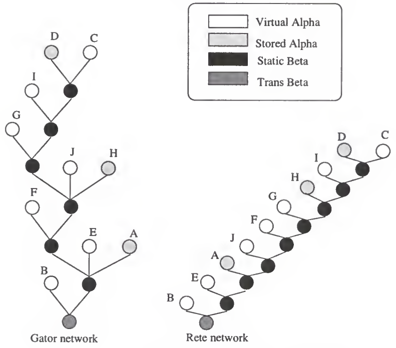


(B) Gator and Rete networks.

Figure 5.11. Optimized Gator and Rete networks for a string type RCG of size 10



(A) Star type RCG with ten relations.



(B) Gator and Rete networks.

Figure 5.12. Optimized Gator and Rete networks for a star type RCG of size 10

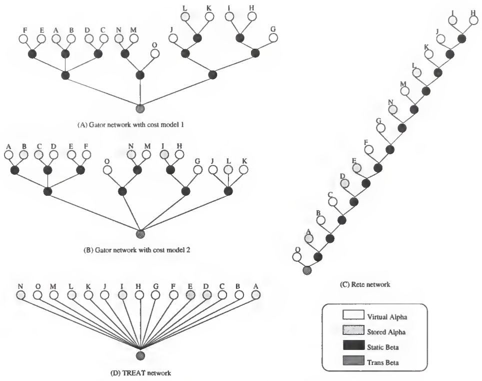


Figure 5.13. Gator, Rete and TREAT networks

optimizer tries to create fewer beta nodes to lower the overall network cost. It was also noticed that the costs of optimized Rete networks in cost model 2 are higher than those of TREAT networks. In case of a TREAT network, there are no beta nodes and hence the cost of the network includes the join processing cost only. Since the frequency distribution is equal, Rete spends more time to check the condition for those relations at the top of the network and hence has a higher overall token processing time than the Gator. The shape of TREAT is balanced but a node in TREAT has more neighbors than that of the Gator. The shape of the Gator is balanced and the tokens entering its leaf nodes perform join operations with fewer nodes than TREAT (because of the beta nodes) and hence has superior performance compared with both Rete and Gator.

In general, it was noticed that the beta nodes in optimized Gator networks have a fan-out of two to four. And, the fan-out of beta nodes in cost model 2 is higher than that of the ones in cost model 1. An important result of this work is that for most cases, including for even update frequency distribution, the optimized Gator network shape is neither pure Rete not TREAT but an intermediate form, with a few beta nodes. The update frequency distribution of relations has a noticeable effect on the shape of the optimized Gator network. For Equal frequency distribution, the shape of the optimized Gator network is balanced with almost equal root-to-leaf path length for all leaf nodes. For step and skew frequencies, the shape of the optimized Gator network is different, with the relations having high update frequencies placed closer to the P-node. Here, the root-to-leaf path lengths were not the same for different leaf nodes, giving the Gator network a more Rete-like shape.

The performance of the optimized Gator network in cost model 1 is significantly better than that of TREAT and reasonably better than that of the optimal Rete. Pattern matching in TREAT involves performing join operations with all the alpha nodes in the network. Gator and Rete take advantage of the pre-computed results in the beta nodes to avoid some of these join operations. Also, the maintenance cost of beta nodes in Gator and Rete is low because of the availability of large buffer space (cost model 1). These factors contribute to the superiority of Gator and Rete over TREAT in cost model 1. Bushy trees allow more flexibility in the shape of a binary tree network and hence the performance of optimized bushy Rete networks would be better than that of left-deep Rete networks and would be even closer to that of optimized Gator networks. The optimizer results show that in a low-buffer environment (in cost model 2), TREAT networks have lower cost than optimal Rete networks. In conclusion, the available buffer space and the update frequency distribution of relations have a significant effect on the optimized Gator network shape.

5.4 Conclusion

This chapter reported the results of an extensive experimental study conducted to study the relative performance of Gator, Rete and TREAT networks. The conclusions of this study are given below.

Experimental results show that Gator performs better than Rete and TREAT in both the cost models. There is no clear winner between Rete and TREAT. In a plentiful-buffer environment, Rete performs better than TREAT while in a low-buffer environment, TREAT outperforms Rete. For the low-buffer environment, the three networks were compared based on the optimizer cost estimates and not on the actual token propagation times.

Experimental results also show that optimized Gator networks normally have a shape which is neither pure Rete nor TREAT, but an intermediate form having a few beta nodes (less than the number of beta nodes in a Rete network). The fan-out of beta nodes in optimized Gator networks depend on the amount of buffer space that is available. In a plentiful-buffer environment, optimized Gator networks are observed to have a fan-out of two to four. In a low-buffer environment, the fan-out is higher.

The update frequency distribution and the amount of available buffer space have a noticeable effect on the shape of optimized Gator networks. But, they do not totally dominate other factors such as data size, rule condition graph shape and predicate selectivity.

Validation of Gator network cost functions have shown that join selectivity is a crucial factor in estimating the cost of a Gator network and that it has a significant effect on the accuracy of the estimated cost of a Gator network.

CHAPTER 6
MULTIPLE RULE OPTIMIZATION

Chapters 2 through 5 discussed the application of discrimination networks (Rete, TREAT & Gator) for doing efficient pattern matching for rules in a database system. The problem of finding an optimal Gator network for a rule was explored, given information about database size, attribute cardinality and update frequency distribution. If there are a number of rules defined in a database in the context of a single application or a set of related applications then it is possible that there is a overlap in the conditions of different rules. This opens the possibility of sharing a discrimination network or a subnetwork of a discrimination network among different rules to reduce the pattern matching time or the rule condition testing time further. Sharing discrimination networks has other advantages also: it saves space and lowers the update processing cost.

Consider the following example which illustrates rules having an overlap in their conditions. The schema for a banking application involving transactions through ATM cards is given below.

ATMCardInfo(ATMCardNo, ssNo)

Depositor(ssNo, name, accountNumber, street, city)

Account(accountNumber, balance)

TransactionReq(trNo, date, ATMMachineNo, ATMCardNo, amount, status)

ATMCardInfo gives information about the ATM cards and their owners. Depositor and Account tables have their usual meaning. TransactionReq records transactions involving an ATM card. Whenever a customer tries to withdraw money using an ATM card, an entry will be created in TransactionReq. Suppose that a bank has the following policy about withdrawing money using an ATM card: an user can always withdraw an amount of money that is less than or equal to the amount in his/her account. When a customer withdraws some amount of money satisfying this condition, that amount will be deducted from that customers account. If the amount of money (the customer is trying to withdraw) is higher than the amount in that customers account, the transaction will be refused and an entry will be made in a log about this transaction. Rules in Figures 6.1 and 6.2 perform this task. Figure 6.3 shows the Gator networks for UpdateAccountRule and AbortTransRule sharing the subnetwork rooted at $\beta 2$.

This chapter explores the problem of optimizing the pattern matching time of a set of rules in a database system. We name this problem *Multiple Rule Optimization* (MRO). It is assumed that Gator networks are used to test the conditions of rules and MRO is defined accordingly below. From now on, the optimal Gator networks of rules obtained by optimizing them individually are referred to as *locally optimal Gator networks*. Generally, merging the locally optimal Gator networks of rules does not give a globally optimal solution.

```
define rule UpdateAccountRule
on append to TransactionReq
from TransactionReq, ATMCardInfo, Depositor, Account
if TransactionReq.ATMCardno = ATMCardInfo.ATMCardno
and ATMCardInfo.ssNo = Depositor.ssNo
and Depositor.accountNumber = Account.accountNumber
and TransactionReq.amount <= Account.amount
then
    do
     /* update the Account table */
     replace Account(amount=amount−TransactionReq.amount);
     /* set the status of the transaction as 'accepted' */
     replace TransactionReq(status='accepted');
    end
```

Figure 6.1. UpdateAccountRule: checks the validity of a transaction and accepts it if it satisfies the desired condition

```
define rule AbortTransRule
on append to TransactionReq
from TransactionReq, ATMCardInfo, Depositor, Account
if TransactionReq.ATMCardNo = ATMCardInfo.ATMCardno
and ATMCardInfo.ssNo = Depositor.ssNo
and Depositor.accountNumber = Account.accountNumber
and TransactionReq.amount > Account.amount
then
    do
     raise event Error(Depositor.name,TransactionReq.ATMMachineNo);
     append to log-table(Depositor.name, TransactionReq.ATMMachineNo,
                         TransactionReq.ATMCardNo, TransactionReq.date,
                         TransactionReq.amount);
     /* set the status of the transaction as 'rejected' */
     replace TransactionReq(status='rejected');
    end
```

Figure 6.2. AbortTransRule: checks the validity of a transaction and rejects it if it does not satisfy the desired condition

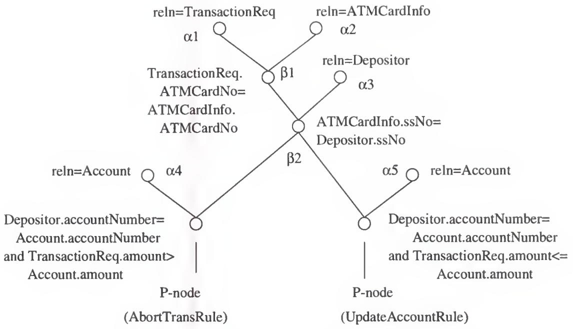


Figure 6.3. Rules sharing a common sub-expression

Problem Definition. Assume that we are given a set of rules $\{R_1, \ldots, R_n\}$ defined on the relations in a database D. Let $GS_i = \{G_{i1}, \ldots, G_{im}\}$ be the set of possible Gator networks that can do pattern matching for a rule R_i, $1 \leq i \leq n$. Multiple Rule Optimization is the problem of finding a Gator network, G_{ij}, for each rule, R_i, $1 \leq i \leq n$, from GS_i, such that the total cost of pattern matching (the total cost of Gator networks for all rules with the cost of shared nodes counted only once) for all the rules is minimal.

There are a lot of similarities between MRO and the problem of Multiple Query Optimization (MQO). The goal of MQO is find a globally optimal access plan for a set of queries. In both the problems, the goal is to improve the total processing time (query processing time in case of queries and pattern matching time in case of rules) of a group of queries (a rule condition can be treated as a query) by taking advantage of the common computation between them. However, we believe that

MRO has the potential to give more benefits than MQO. The reason is that, in a database system, the rule conditions are checked for activation on every update to the database and this process is repeated until the rules are deactivated. Hence, the benefits due to MRO are going to be substantial. If rules are processed by brute force approach then MRO is essentially the same as MQO. Also, MRO can be used for view materialization, which has been a hot topic lately.

The rest of this chapter is organized as follows: Section 6.1 presents a brief survey of related work in multiple rule optimization and multiple query optimization. Section 6.2 gives a brief description of the proposed approach to solve the multiple rule optimization problem. Section 6.3 presents an algorithm to find common subexpressions among the RCGs of rules. Section 6.4 gives a detailed version of the proposed approach. A new search strategy based on randomized algorithms is also presented in this section. Section 6.5 discusses the simulator that has been developed to generate the test data for the proposed search algorithm. Section 6.6 concludes the chapter with a discussion of experimental results.

6.1 Survey of Related Work

6.1.1 Related Work on Multiple Rule Optimization

Ishida [25] presents an approach for doing Multiple Rule Optimization in production systems. Here, Rete networks are used for doing pattern matching for rules. Each rule is optimized separately by using a dynamic programming based algorithm.

While optimizing each rule separately, the approach allows for sharing of join expressions among rules. A sub-network of a Rete network that does pattern matching for a join expression is called a join structure.

The algorithm works as follows. First, join structures of size 1 (alpha nodes) are created. Join structures of size i are constructed by joining the already created join structures of size ranging from 1 to $i - 1$. After creating all possible join structures, the one (which is complete) with the lowest cost is selected as the optimal one. Let JE_i represent the join expression of a join structure JS_i. During optimization, the cost of a join structure, JS_i, is calculated based on the following heuristic: all the rules that contain JE_i in their rule condition are going to share it. The cost of JS_i is calculated by dividing its actual cost by the number of rules that can share it. While optimizing the other rules (that can share JS_i), the cost of JS_i is made as 0. The cost of JS_i is significantly lowered in order to encourage the optimizer to choose shared join structures. Finally, a join structure is shared among those rules that have selected it as part of their optimal join structures. Since the costs are assigned to join structures based on heuristics the global solution does not guarantee optimality.

6.1.2 Related Work on Multiple Query Optimization

Finkelstein [10] suggests an approach for computing the result of a query by processing the materialized results of previous queries. He presents an algorithm to decide whether a given query forms a subexpression of another query. Park and Segev [26] present a dynamic programming algorithm to solve the Multiple Query Optimization problem.

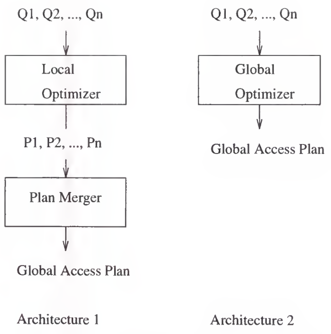


Figure 6.4. Architectures proposed by Sellis

Sellis [45] proposes two architectures for Multiple Query Optimization. They are shown in Figure 6.4. Architecture 1 attempts to achieve MQO with minimal changes to the existing query optimizer. The Local query optimizer generates an optimal plan for each query. The plan merger module generates a global access plan by combining the locally optimal plans. A hierarchy of algorithms of varying complexity is presented to identify the common subexpressions among the locally optimal plans of queries. Architecture 2 requires extensive changes to the existing query optimizer. Here, the search space is extended to include the non-optimal plans of queries also. The Global Optimizer module uses an A*-algorithm to find the optimal global access plan.

Shim [46] presents a heuristic algorithm for MQO which is an extension of the A*-algorithm given by Sellis [45]. Here, the value of the heuristic function for a given node in the search tree is calculated dynamically during the search process and hence

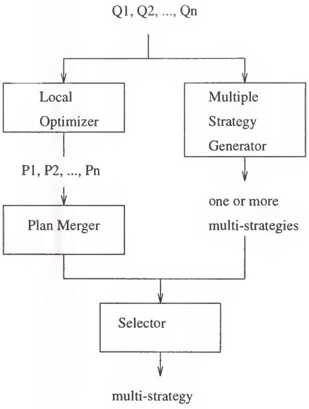


Figure 6.5. Partitioned approach

is more informed than the one given in Sellis[45]. This algorithm handles implication of queries also.

Chakravarthy [9] proposes a partitioned approach for Multiple Query Optimization, shown in Figure 6.5. Here, the search space is divided into two spaces and a solution is obtained by picking the best global plan out of these two spaces. In partition 1, locally optimal plans of queries are combined to generate a global plan. In partition 2, a global plan is generated by combining the sub-optimal plans of queries. Heuristics are suggested to constrain the search space in partition 2.

6.2 Description of the Proposed Approach

This section presents a new approach to solve the Multiple Rule Optimization problem. Figure 6.6 illustrates the proposed architecture. The existing local Gator network optimizer is used to generate a locally optimal Gator network for each rule. In addition, for each rule, a set of sub-optimal Gator networks are generated by using a slightly modified version of the existing local optimizer. The number of sub-optimal Gator networks generated for a rule depends on the number of shared expressions between that rule and the other rules in the system. Finally, search is done in the search space consisting of all the Gator networks of all the rules generated as above and the best global solution is generated. The highlight of this approach is that only those sub-optimal networks that have the potential to contribute to the global solution participate in the search process. The sub-optimal Gator networks that are generated by the modified local optimizer are called the *locally optimal sharable* Gator networks.

A *sharable* Gator network of a rule R_i for an expression e_i is a Gator network G_i of R_i satisfying the following condition: all the nodes (both alpha and beta) in G_i that materialize e_i must be descendants of some unique beta node in the network and no other node must be a descendant of that beta node. There can be more than one *sharable* Gator network for a rule for a given expression. For example, in Figure 6.7, for the rule Rule3, Gator networks G1 and G2 are in a sharable form for the expression {R1.a=const1 and R1.b=R2.b and R2.c=R3.c } while the Gator network G3 is not in a sharable form. An *optimal sharable* Gator network of a rule

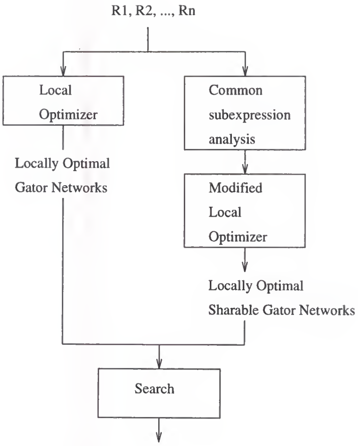
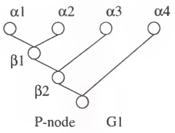
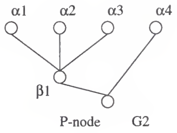
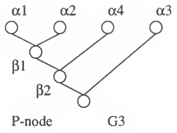

R1, R2, ..., Rn
Local Optimizer
Common subexpression analysis
Locally Optimal Gator Networks
Modified Local Optimizer
Locally Optimal Sharable Gator Networks
Search

Figure 6.6. The proposed approach





```
define rule   Rule3
if R1.a = const1
and R4.b = const2
and R1.b = R2.b
and R2.c = R3.c
and R2.d = R4.d
then Action
```



LEGEND:

α1 –	alpha node for relation R1
α2 –	alpha node for relation R2
α3 –	alpha node for relation R3
α4 –	alpha node for relation R4

Figure 6.7. Sharable Gator networks

for an expression is a *sharable* Gator network of that rule for that expression with minimum cost.

The main idea behind this approach is the following: in order for a rule to share an expression with other rules, its Gator network must be in a *sharable* form. An *optimal sharable* Gator network is the best (or the lowest cost) Gator network that is available for a rule in order to share an expression with other rules. If the *optimal sharable* Gator networks of all the rules for all their corresponding common sub-expressions can be found then a globally optimal solution can be obtained by doing a search only among the collection of *optimal* and *optimal sharable* Gator networks of rules.

The condition of a rule can be treated as a query and a query graph constructed for it is called a *Rule Condition Graph* (RCG). The RCG's of rules are analyzed for common sub-expressions. In this analysis, subsumption of expressions is considered only at the selection level. At the join level, only equivalence of expressions is considered. The RCG's of rules are analyzed to find common join-clusters of maximum size among rules.

Let's assume that there are n expressions that are common between a rule R_i and the other rules in the system. In the optimal global solution, R_i may share any one of these common expressions or any combination of these with the other rules. There are 2^n ways that R_i can share n expressions with the other rules in the system. However, generating 2^n optimal sharable Gator networks for a rule is unacceptable. The sharing possibilities are reduced by using some constraints and heuristics. The details of the proposed heuristics are given in section 6.4.

Finally, a search strategy based on randomized algorithms is presented to find an optimal global strategy among the locally optimal and locally optimal sharable Gator networks of rules.

Section 6.3 presents an algorithm to find common sub-expressions among the RCGs of rules. Section 6.4 gives a detailed version of the proposed approach.

6.3 Finding Common Sub-expressions

This section presents an algorithm to detect common sub-expressions among the RCG's of rules. Common sub-expressions allow an alpha node or a subnetwork of a Gator network to be shared among the Gator networks of different rules. In this analysis, subsumption of expressions is considered only at the selection level. At the join level only equivalence of expressions is considered.

The following definitions are from Sellis [45]. A *selection predicate* is a predicate of the form $R.A$ *op cons* where R is a relation, A a field of R, $op \in \{=, \neq, <, \leq, >, \geq\}$ and *cons* some constant. A *join predicate* is a predicate of the form $R1.A = R2.B$, where $R1$ and $R2$ are relations, A and B are fields of $R1$ and $R2$ respectively.

For simplicity, it is assumed that a rule condition has the following property: it is a conjunction of selection and join predicates only. The following two terms are defined on a rule condition: In a rule condition, an *S-expression* of a relation is a conjunction of all the selection predicates defined over it. A *Join Expression* (or *J-expr*) of a set of relations R_{set} is a conjunction of S-expressions of all the relations in R_{set} and join predicates between all possible pairs of relations in R_{set}.

An S-expression essentially represents the condition of an alpha node and a J-expression represents the conditions of all the alpha and beta nodes in a subnetwork of a Gator network.

The following definition about the implication of S-expressions is from Sellis [45]. Let SE_1 and SE_2 be two S-expressions. SE_1 *implies* SE_2 ($SE_1 \Rightarrow SE_2$) iff SE_1 is a conjunction of selection predicates on a relation R and on attributes $A_1, A_2, \ldots, A_k$ and SE_2 is a conjunction of selection predicates on the same relation R and on attributes $A_1, A_2, \ldots, A_l$ with $l \leq k$ and the result of evaluating SE_1 is a subset of the result of evaluating SE_2 for all possible values of R. SE_1 is *identical* to SE_2 iff $SE_1 \Rightarrow SE_2$ and $SE_2 \Rightarrow SE_1$.

Join Expressions JE_1 and JE_2 of a relation set, R_{set}, are *identical* iff: 1. JE_1 and JE_2 contain identical S-expressions for all the relations in R_{set} and 2. JE_1 and JE_2 contain identical join predicates between all possible pairs of relations in R_{set}.

A join predicate $R_1.a_1 = R_2.b_2$ is *identical* to $R_{11}.a_{11} = R_{22}.b_{22}$ iff $R_1 = R_{11}$, $R_2 = R_{22}$, $a_1 = a_{11}$ and $b_2 = b_{22}$.

For example, in Rule3 in Figure 6.7, the S-expr of relation R1 is given by {R1.a=-const1} and J-expr of relations $\{R_1, R_2\}$ is given by {R1.a=const1 and R1.b=R2.b}.

Let SE_1 and SE_2 be the S-expressions of a relation in two rules. If SE_1 is identical to SE_2 then an alpha node created with the selection condition as SE_1 can be shared between the Gator networks of the two rules. If SE_1 implies SE_2, the corresponding alpha nodes can be shared in the following way: an alpha node α_2 can be created with the selection condition as SE_2 and the second alpha node α_1, for SE_1, can use α_2 to derive its tuples. The selection condition of α_1 can be made to be SE_2/SE_1 (the remainder of SE_2 that differs from SE_1). If a set of relations has identical J-expressions in different rules then the subnetwork that is materializing (or doing pattern matching for) that J-expression can be shared between them.

S-expressions of a relation belonging to different rules are said to be *common* if they are either identical or one implies the other. J-expressions of a set of relations belonging to different rules are said to be *common* if they are identical.

6.3.1 Finding Common S-expressions

The table driven approach given in [10] can be used to detect identical and implied S-expressions among rules. Please refer to [10] for more details.

6.3.2 Finding Common J-expressions

If a join expression of a relation set, R_{set}, is common among a set of rules, RL, then a join expression of any subset of R_{set} is also going to be common among RL. In order to reduce the sharing possibilities, only a join expression of maximum size is allowed to be shared among the Gator networks of different rules. The size of a join expression is the number of relations in its relation set. *Eligible* J-expressions are the common J-expressions satisfying this condition.

A join expression of a set of relations, R_{set}, which is common among rules RL is said to be *eligible* if there exists no join expression which is common among the same set of rules RL and whose relation set is a superset of R_{set}.

Only eligible common J-expressions are considered for sharing. An eligible common J-expression gives a common join-cluster of maximum size.

The following algorithm finds all eligible common J-expressions among rules. In addition, the algorithm constructs the following: RE-list for each rule and ER-list for each common J-expression. The RE-list of a rule is the list of all J-expressions that are common between that rule and the other rules in the system. The ER-list of a J-expression is the list of all rules that contain it as part of their RCG. In the following algorithm, JE^n refers to eligible common J-expressions of size n.

Input: A set of rules and their rule condition graphs.

Output: e_{list}, List of all eligible common J-expressions between the given set of rules.

```
1. n=2;
   Find all common J-expressions of size 2 among all rules and append them to $e_{list}$.
```

```
    For each rule, append the J-expr's that are common between that rule and the
    other rules to its RE-list.
    For each J-expr, fill its ER-list.
2. While ( true ) do
        $n = n + 1$; // each iteration looks for J-expressions of size $n$
        For each J-expr, $e_i$, in $JE^{n-1}$ do
            For each J-expr, $e_j$, in $JE^2$ do
                If $e_i$ and $e_j$ are connected
                Then
                    $e_k$ = Form a new J-expr($e_i$, $e_j$)
                    If $e_k$ does not already exist in $JE^n$
                    Then
                        Append $e_k$ to $e_{list}$.
                        ER-list of $e_k$ = ER-list of $e_i \cap$ ER-list of $e_j$
                        Append $e_k$ to the RE-list of each rule in its ER-list
                        //Find the J-expr's in $JE^{n-1}$ that are not going to be
                        //eligible due to $e_k$
                        For each J-expr, $e_l$, in $JE^{n-1}$ do
                            If the relation set of $e_l$ is a subset of the relation set
                            of $e_k$ and if the ER-list of $e_l$ is same as that of $e_k$
                            Then
                                Mark $e_l$ as not eligible.
                    Delete $e_l$ from $e_{list}$
                End For
            End If
        End If
    End For
  End For
End While
```

It can be noticed that the above algorithm is similar to the Interleaved Execution (IE) algorithm given in [45]. Even though the above algorithm performs more tasks than the IE algorithm, such as maintaining ER-lists for CSEs, RE-lists for rules and checking the eligibility criterion for CSEs, its aymptotic time complexity is the same as that of the IE algorithm.

The complexity of Step 1 is in the order of $\prod_{i=1}^{m} N_i$, where N_i is the number of vertices in the RCG of a rule R_i and m is the number of rules. In Step 2, the number of times K the while loop is executed depends on the size of common sub-expressions and, in the worst case, is equivalent to the size of the RCG that has the highest number of vertices. Hence, the total complexity is $K \cdot \prod_{i=1}^{m} N_i$.

This concludes the discussion of the algorithm to find common sub-expressions among the RCGs of rules. The details of the proposed approach to solve the multiple rule optimization problem which makes use of the algorithm presented above, are given next.

6.4 The Proposed Approach for MRO

The previous section presents algorithms to find common-sub expressions (common S-expressions and J-expressions) among the RCGs of rules.This section discusses the proposed approach to find an optimal sharing of common J-expressions among rules. A common S-expression allows an alpha node to be shared and a common J-expression allows an entire subnetwork of a Gator network to be shared among rules. For all the relations that have common S-expressions, the corresponding alpha nodes are shared. The following discussion concentrates on the problem of finding an optimal sharing of J-expressions (that is, sharing of β nodes) among the Gator networks of different rules.

From now on, the term *contiguous common sub-expression (CCSE)* is used to refer to an eligible common J-expression, the term *composite common sub-expression* to refer to a collection of contiguous common sub-expressions and the term *CSE* to refer to both a contiguous CSE and a composite CSE, when the distinction between them is not important. Contiguous CSEs and Composite CSEs are explained with the help of an example next. Figure 6.8 shows the RCGs of three rules R1, R2 and R3. The numbers $1, 2, \ldots, 8$ in the diagram represent tuple variable nodes. The contiguous CSE, C1, is common between R1 and R2 and the contiguous CSE, C2, is common between R1 and R3. A composite CSE is a collection of CCSEs. {C1, C2} is a composite CSE of R1. There are three ways that R1 can share its CSEs with R2 and R3. R1 can share C1 with R2 or C2 with R3 or both C1 and C2 with R2 and R3

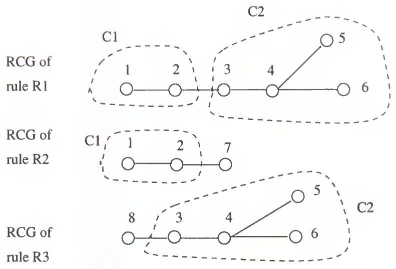


Figure 6.8. Contiguous CSEs and composite CSEs

respectively. In the third case, we say that R1 shares the composite CSE {C1,C2} with R2 and R3.

The details of the proposed approach to find an optimal sharing of common J-expressions among a set of rules, are given next.

Input: A set of rules and their Rule Condition Graphs. Let $R_1, \ldots, R_k$ be the set of rules and let $RCG_1, \ldots, RCG_k$ be their respective Rule Condition Graphs.

Step 1: Find all the contiguous common sub-expressions (CCSEs) among $R_1, \ldots, R_k$ using the algorithm given in Section 6.3.2. Contiguous common sub-expressions are the common J-expressions of maximum size.

Let CSE_{list} be the list of all CCSEs found in step 1.

Step 2: The RE-list of a rule is the list of all expressions that are common between that rule and the other rules in the system. The ER-list of a CCSE is the list of all rules that contain this expression as part of their RCG. The *degree* of a CCSE is the number of rules in its ER-list.

Construct RE-list for each rule and ER-list for each CCSE in CSE_{list}, using the algorithm in Section 6.3.2.

Step 3: For each rule, generate a *locally optimal* Gator network and a set of *locally optimal sharable* Gator networks. The details are given below.

Let R_i be a rule in the system and let n be the number of CCSEs in the RE-List of R_i. In the optimal global solution, R_i may share any one of these CCSEs or any combination of these with the other rules. In other words, R_i may share any of the contiguous CSEs or any of the composite CSEs with the other rules. There are 2^n ways that R_i can share n CCSEs with the other rules in the system. However, generating 2^n optimal sharable Gator networks for a rule is unacceptable.

The sharing possibilities are being reduced by using heuristics. For a rule having n CCSEs, there are $2^n - n - 1$ possible composite CSEs that can be generated. This is obtained by counting the number of composite CSEs of size 2, size 3 and so on upto size n ($\binom{n}{2} + \binom{n}{3} + \ldots + \binom{n}{n} = 2^n - n - 1$). The search space is being reduced by selecting only a subset of all possible composite CSEs for sharing among the rules. All the contiguous CSEs are allowed to be shared among the rules. The selected composite CSEs have to be such that they should not explode the search space and at the same time they should allow as much sharing as possible. The heuristics that are explored for selecting the composite CSEs are given next. In the following discussion, the cost of a CSE refers to the cost of the Gator network that does pattern matching for that CSE.

Pick-2^k: Let n be the number of CCSEs in the RE-list of a rule R_i. From the n CCSEs, select k potential CCSEs and generate all combinations of one to k of these k CCSEs. This way, $2^k - k - 1$ composite CSEs are generated for R_i. k is a parameter whose value can be set by experimentation or depending upon the time constraints under which the algorithm should run. k may be a fixed number for all the rules in the system or its value may be set independently for each rule based on the costs of CCSEs of that rule (for example, select all CCSEs whose costs are within 50% of the highest cost CCSE of that rule).

Three different criteria are used for selecting the k CCSEs. They are given below.

Sort On Cost: Sort CCSEs in decreasing order based on their cost and select k CCSEs starting from the highest cost CCSE in the sorted order. This criterion was selected based on the assumption that sharing the CCSEs having higher cost is more beneficial than the ones having lower cost. If the cost distribution of the CCSEs of a rule is skewed then this criterion allows generating a solution of very good quality.

Sort on Cost*NoOfRules: For each CCSE, compute $(Cost * NoOfRules)$ where *cost* refers to the cost of a CCSE and $noOfRules$ gives the number of rules sharing that CCSE. Sort CCSEs based on this cost function in decreasing order and select k CSEs starting from the highest cost CCSE in the sorted order. The motivation behind choosing the value *(Cost*NoOfRules)* is that for some CCSEs, even though their cost is low, the number of rules sharing them may be high. In those cases, the value *(Cost*NoOfRules)* may be a better measure to select the CCSEs than just *Cost*.

Sort On Cost/NoOfRules: For each CCSE, compute the value (*Cost / NoOf-Rules*) where *Cost* refers to the cost of a CCSE and $NoOfRules$ gives the number of rules sharing that CCSE. Sort CCSEs in decreasing order based on this cost function and select k CCSEs starting from the highest cost CCSE in the sorted order. This criterion is useful when a CCSE is shared by only a subset of all the possible rules that can share that CCSE. Also, if optimal solutions share expensive CSEs that are common among only a small number of rules, this heuristic could help find those solutions. Since it is plausible this heuristic may help in some situations, we decided to try it as one possibility in the simulation.

Even though, the number of composite CSEs generated in Pick-2^k increases exponentially with k, by choosing a smaller value of k, we hope to recognize as much sharing as possible without significantly increasing the search space. Again, since the number of generated composite CSEs increases exponentially, it is not possible to choose a high value for k.

Pick-k^2: For each rule, select k contiguous CSEs as in previous heuristic but generate the composite CSEs in the following way: First, generate $(k-1)$ composite CSEs of size 2 by combining the adjacent CCSEs in the sorted list. Next, generate $(k-2)$ composite CSEs of size 3 by combining the three adjacent CSEs in the sorted list and so on until a composite CSE of size k is generated by combining all the CSEs in the sorted list. For example, let $1, 2, 3, 4$ be the CCSEs that are selected (in that order) from the sorted list of CCSEs of a rule. The composite CSEs generated are: $12, 23, 34, 123, 234$ and 1234. Here, the number of composite CSEs generated is $n + k(k-1)/2$. The goal is to generate fewer CSEs than the heuristic Pick-2^k ($O(k^2)$ as opposed to $O(2^k)$). This will allow selecting a higher value for k than the one allowed by the heuristic Pick-2^k.

Pick-k: Select k contiguous CSEs as in previous heuristics but select the composite CSEs in the following way: generate one composite CSE of size 2 by combining the first two CCSEs in the sorted list, one composite CSE of size 3 by combining the first three CCSEs and so on until a composite CSE of size k is generated. For example, if $1, 2, 3$ are the selected CCSEs (in that order) from the sorted CCSE list of a rule, then the composite CSEs generated are: $12, 123$ and 1234.

The goal in heuristics Pick-k^2 and Pick-k is to generate fewer composite CSEs than the ones in Pick-2^k for a chosen value of k ($O(k^2)$ and $O(k)$ as opposed to $O(2^k)$). This will allow choosing a higher value for k (and hence include more CCSEs in the searching process) than the one allowed by Pick-2^k.

For each rule, composite CSEs are generated as explained above and are appended to their RE-lists. For each rule, a *locally optimal* Gator network is generated by using an existing single-rule optimizer. By using a slightly modified version of the existing optimizer, a *locally optimal sharable* Gator network is generated for each CSE in the RE-list of a rule. Hence, for a rule with n common sub-expressions, $n + f(k)$ ($f(k) = O(2^k)$, $O(k^2)$, or $O(k)$), depending upon the heuristics used) optimal sharable Gator networks are generated.

The following notation is used below: the *optimal sharable* Gator network of a rule R_i for an expression e_i is denoted by $G_{r_i e_i}$ and the *optimal* Gator network of R_i is denoted by $G_{r_i O}$. The set of *optimal* and *optimal sharable* Gator networks generated for a rule R_i is denoted by GN_i.

Step 4: MRO is the problem of finding Gator networks for a set of rules such that the total cost of pattern matching for all the rules is minimal. Steps 1 and 2 find common sub-expressions among rules. Step 3 generates locally optimal and locally optimal sharable Gator networks for rules based on the CSEs generated in Step 1. The last step in the proposed approach, as shown in Figure 6.6, involves performing search among the locally optimal and the locally optimal sharable Gator networks of rules.

The problem of finding an optimal sharing of CSEs among Gator networks is an NP-hard problem. It is equivalent to the problem of finding a minimum-weight subgraph of an AND/OR graph [33, 42]. Hence, finding a truly optimal solution is not a realistic goal.

Randomized algorithms have been successfully applied to various combinatorial optimization problems [1, 59]. Chapter 4 reports the application of randomized algorithms for the Gator network optimization problem. The advantages of these algorithms include simplicity, flexibility, minimal memory requirement and in many cases, high-quality results. Motivated by the above mentioned reasons, we applied randomized algorithms for the MRO search problem. The different randomized algorithms that were explored in the context of query optimization and Gator network optimization were II, SA and TPO. From our past experience in applying these algorithms to the Gator network optimization problem, II required very little tuning effort compared to SA and TPO and the performance of II was close to that of others. Because of these reasons, only the II strategy was considered for the MRO search problem.

A general description of randomized algorithms is given next. The details of problem specific parameters are given after that.

6.4.1 Randomized Algorithms for the Search Problem

Each solution to a combinatorial optimization problem can be viewed as a *state* in a state space. Each state in the state space is associated with a cost, calculated by using a problem-specific cost function. The aim of an optimization algorithm is to find a state with the lowest cost in the state space. A *move* or a *local change operator* is an operation applied to a state to obtain another state. All the states that can be reached from a state in one move are called the *neighbors* of that state. A state is called a *local minimum* if the cost of the state is less than that of all its neighbor states. A *global minimum* is the state with the lowest cost in the state space. A move is called a *downhill move* if the cost of the new state obtained by applying a move is less than that of the current state; or if the cost of the new state is higher, it is called an *uphill move.* A *plateau* consists of adjacent states with the same cost.

Iterative Improvement

Iterative Improvement [36, 1] is a local search algorithm that comes under the class of heuristic or approximation algorithms.

```
procedure II()
{
   minState = random state;
   while not (stopping_condition())
   {
      S = random state
      while not(local_minimum(S))
      {
         S' = random state in neighbors(S)
         if cost(S') < cost(S)
             S = S'
      }
      if cost(S) < cost(minState)
          minState = S
   }
   return(minState)
}
```

Figure 6.9. Iterative Improvement

The generic algorithm is shown in Figure 6.9. In each iteration of the outer loop, a random state is generated and a local search is initiated with the generated random state as the start state. The local search process (the inner loop) starts at a given state and applies downhill moves repeatedly until a local minimum is reached. The number of iterations of the outer loop (i.e. the number of local minimum states examined) is controlled by the *stopping criterion.* The output of II is the local minimum state with the lowest cost.

6.4.2 Problem Specific Parameters

State Space

Each solution to the MRO problem with k rules is represented by a k-ary vector, as shown below. The first element in the solution vector, G_1, is the Gator network chosen for rule R_1, the second element, G_2, is the network chosen for rule R_2 and so on.

$$< G_1, G_2, ..., G_k >$$

Let cse_1,...,cse_n be the CSEs that are shared among the Gator networks in a solution s_i and let a CSE cse_i be shared among m_i ($2 \leq m_i \leq k$) Gator networks in s_i. The cost of s_i is defined as:

$$cost(s_i) = \sum_{i=1}^{k} cost(G_i) - \sum_{i=1}^{n} (m_i - 1) * cost(cse_i)$$

The goal of the Multiple Rule Optimization problem is to find a solution with minimum cost. A solution with minimum cost is one that allows maximum beneficial sharing among the rules.

Each Gator network in a solution is associated with a *tag* that gives information about the CSE shared by that Gator network with the Gator networks of other rules in that solution. The format of a tag is given below.

$$< cse_i,\ NoOfRules,\ ListOfRules >$$

Let the tag belong to a Gator network G_i. The tag specifies that G_i shares the CSE cse_i with the Gator networks of rules given by *ListOfRules*. *NoOfRules* gives the number of rules in *ListOfRules*. The ER-list of a CSE gives the list of all the rules in the system that can share that CSE whereas the *ListofRules* in the tag of a rule in a solution gives the list of those rules that are sharing that CSE in that particular solution.

Constructing a Random Solution

Let R_{list} be the set of all the rules in the system and let E_{list} be the set of all the CSEs in the system. Let RE_i be the RE-list of a rule R_i and let ER_i be the ER-list of a CSE e_i.

In a solution, a rule may share a contiguous CSE or a composite CSE with other rules. A composite CSE c_i is called an *extension* of the composite CSE c_j if the set of contiguous CSEs that form c_i is a superset of or equivalent to the contiguous CSEs that form c_j.

While constructing a random solution, a temporary variable, $tempVar_i$, is associated with each rule, R_i. The temporary variable associated with a rule gives the CSE which that rule is sharing with others in the solution. $tempVar_i$ is initially set to NULL for all the rules. Temporary variables are introduced in order to keep track of the CSEs shared by rules and are utilized in creating tags for Gator networks in the generated solution.

A rule R_i is said to be *eligible* to share a CSE e_i with other rules if it has a CSE in its RE-list which is an extension of $(tempVar_i \cup e_i)$ where $tempVar_i$ is the temporary variable of R_i. While constructing the random solution, a list *committedRuleList* is maintained which gives the list of rules for which we have selected the Gator networks. Initially committedRuleList is set to NULL.

The reason for introducing the *eligibility* criterion is the following: Let the RE-list of

The procedure to construct a random solution is given below:

```
While $R_{list}$ is not empty do
    1. Select a rule
            Select a rule $R_i$ randomly from $R_{list}$.
    2. Select a CSE of $R_i$
            $RE_i$ gives the list of all the CSEs of $R_i$.
            If $tempVar_i$ is Null
                    Select a CSE randomly from $RE_i$.
            Else
                    Find all the CSEs in $RE_i$ which are extensions of
                    $tempVar_i$ and select a CSE randomly from those.
            Let $e_i$ be the selected CSE.
    Now, pick those rules which are going to share $e_i$ with $R_i$.
    3. If $e_i$ is a contiguous CSE
            $ER_i$ of $e_i$ gives the list of all rules that can share $e_i$.
            Find all the rules in $ER_i$ which are eligible to share $e_i$ with $R_i$
            and which are not in committedRuleList and pick a set of rules
            randomly from those.
            For each of the selected rules, assign
                    $tempVar_i$=($tempVar_i \cup e_i$).
```

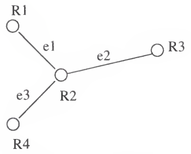


Figure 6.10. A set of rules and their common sub-expressions

```
4. If $e_i$ is a composite CSE
        For each contiguous CSE, $s_i$, in $e_i$ do:
                Find all the rules in $ER_i$ which are eligible to share
                $s_i$ with $R_i$ and which are not in committedRuleList
                and pick a set of rules randomly from those.
                For each of the selected rules, assign
                    $tempVar_i$=($tempVar_i \cup s_i$).
5. If there is no CSE to choose for $R_i$ (because the rules with which $R_i$ has
   common CSEs have already been associated with Gator networks in the
   previous iterations) and its tempVar is NULL, then select $G_{r_iO}$ for $R_i$
   and set its tag to Null.
6. Create a tag for $R_i$.
7. Delete $R_i$ from $R_{list}$ and append it to committedRuleList.
End While
```

The above algorithm is explained with the help of an example below. Figure 6.10 shows a set of rules and the CSEs that are shared by them. R1 to R4 are rules and e1 to e3 are the shared CSEs. The following is a walk-through of the algorithm.

Rule	RE-list
R1	{e1}
R2	{e1, e2, e3, e1e3 }
R3	{e2}
R4	{e3}

R_{list} = {R1,R2,R3,R4} and committedRuleList = NULL

Iteration 1:

Step 1. Select a rule from R_{list}: R1

Step 2. Select a CSE of R1:

tempVar of R1 = NULL

RE-list (CSE list) of R1 = {e1}

Select a CSE from the RE-list list of R1: e1

Step 3. Pick those rules which are going to share e1 with R1:

e1 is a contiguous CSE.

ER-list of e1 = {R1,R2}

Rules (from the ER-list of e1) which are eligible to share e1

with R1 = {R2}

Rules selected (to share e1 with R1) from the above list = {R2}

Update the tempVar of R2. tempVar of R2 = e1

The tempVar of R2 indicate that R2 is *currently* sharing e1 with R1.

Step 6. Create a tag for R1.

Step 7. Delete R1 from R_{list} and append it to committedRuleList

Now, R1 is going to share e1 with R2. In the next iterations, R2 is eligible to share only those composite CSEs which are extensions of e1 (since it is committed to share e1 with R1 in this iteration). This explains the motivation behind introducing the eligibility criterion.

Iteration 2:

R_{list} = {R2,R3,R4}, committedRuleList = {R1}

Step 1. Select a rule from R_{list}: R4

Step 2. Select a CSE of R4:

tempVar of R4 = NULL

RE-list of R4 = {e3}

Select a CSE from the RE-list list of R4: e3

Step 3. Pick those rules which are going to share e3 with R4:

e3 is a contiguous CSE.

ER-list of e3 = {R2,R4}

Rules (from the ER-list of e3) which are eligible to share e3 with R4 = {R2}

R2 is eligible because R2 has a CSE (e1e3) which is an extension of its tempVar (which is e1) and e3.

Rules selected to share e3 with R4 = {R2}

Now, R4 is going to share e3 with R2. R2 is sharing e1 with R1 and e3 with R4.

Update the tempVar of R2. tempVar of R2 = e1e3

Step 6. Create a tag for R4.

Step 7. Delete R4 from R_{list} and append it to committedRuleList.

Iteration 3:

R_{list} = {R2,R3}, committedRuleList = {R1, R4}

Step 1. Select a rule from R_{list} = R3

Step 2. Select a CSE of R3:

tempVar of R3 = NULL

RE-list of R3 = {e2}

Select a CSE from the RE-list list of R3: e2

Step 3. Pick those rules which are going to share e2 with R3:

e2 is a contiguous CSE.

ER-list of e2 = {R2,R3}

Rules (from the ER-list of e2) which are eligible to share e2 with R3 = NULL

R2 is not eligible because it has no CSE which is an extension of its tempVar (which is e1e3) and e2.

Rules selected to share e2 with R3 = NULL

Step 5. R3 is not going to share any CSE with other rules. Hence, assign its optimal Gator network to it.

Step 6. Create a tag for R3.

Step 7. Delete R3 from R_{list} and append it to committedRuleList.

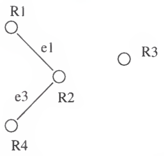


Figure 6.11. The generated solution

Iteration 4:

R_{list} = R2, committedRuleList = {R1, R4, R3}

Step 1. Select a rule from R_{list} = R2

Step 2. Select a CSE of R2:

tempVar of R2 = e1e3

The CSEs in the RE-list of R2 which are extensions of e1e3 = NULL

Select a CSE from the above list: NULL

Hence, R2 is going to e1 with R1 and e3 with R4.

Step 6. Delete R2 from R_{list} and append it to committedRuleList

Step 7. Create a tag for R2.

This completes the tracing of the algorithm. The solution generated is shown in Figure 6.11.

Local change operator or Neighbors Function

Swap Gator networks (or Exchange a CSE or Exchange CSEs). The Gator network of a randomly chosen rule is replaced with one of the other choices possible for that rule.

Replacing the Gator network of a rule in a solution might affect the Gator networks that are associated with the other rules in the solution. The rules that will be affected are the ones with which the selected rule is sharing a CSE in the current solution and the rules which are sharing CSEs (in the current solution) with those with whom the selected rule is going to share a CSE in the new generated solution. Among all the rules affected, we try to find new common CSEs or assign optimal Gator networks (in which case those rules do not share any CSEs with other rules) to them.

This is illustrated with the help of an example next. In Figure 6.12, the nodes denoted by R1, R2, R3, R4, R5, R6 and R7 represent rules and the edges denoted by e1, e2, e3 and e4 represent the contiguous CSEs that are shared among these rules. In the solution shown in the figure, R2 shares e1 with R1 and e2 with R3, R4 shares e3 with R5 and R6 shares e4 with R7. In a new solution, if R2 shares e5 with R6 (shown by dashed line in the figure) then the rules affected are: 1. R1 and R3, which are sharing a CSE with R2 and 2. R7, which is sharing a CSE in the current solution with R6. R6 is going to share e5 with R2 in the new solution and hence the rules that are sharing a CSE with it in the current solution (in this case, R7) are also affected. After swapping the Gator networks of the selected rule, the local change operator tries to find new common CSEs among the affected rules (here, R1, R3 and R7). Figure 6.13 shows the status of the new solution after swapping the Gator networks of R2 and R6.

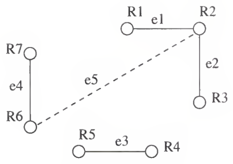


Figure 6.12. A random solution (before applying the local change operator)

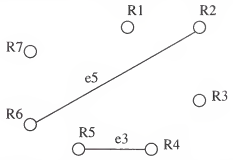


Figure 6.13. The new solution (obtained after applying the local change operator)

The details of the operator are given next. Let R_i be the rule selected, let G_{r,e_i} be the Gator network of R_i in the current solution. $G_{r_j e_j}$ is the *optimal sharable* Gator network of rule R_j for the CSE e_j. Let $< e_i, k, List_i >$ be the current tag of R_i.

1. Pick a CSE (contiguous or composite) randomly from the RE-list of R_i. Let e_j be the selected CSE.
2. Select the rules which are going to share e_j with R_i in the new solution.

   2.1 If e_j is a contiguous CSE

      Rules are selected from the ER-list, er_j, of e_j. Let N be the number of rules (other than R_i) in er_j.

      Pick a set of m $(1 \leq m \leq N)$ rules randomly from er_j.

```
    2.2 If $e_j$ is a composite CSE
            For each of the contiguous CSEs, $c_j$, in $e_j$ do
                Select rules from the ER-list of $c_j$ as in step 2.1
    Let $List_j$ be the list of selected rules and let $j$ be the number of rules
    in $List_j$.
3. Find the rules that are affected because of the new sharing between $R_i$ and
   the rules in $List_j$. Place the affected rules in a list called affectedRuleList.
    3.1 All the rules in the tag of $R_i$ (in the old solution) are affected.
        Append the rules in $List_i$ to affectedRuleList.
    3.2 All the rules which were sharing CSEs with the rules in $List_j$ in
        the old solution are affected.
        3.2.1 For each rule, $R_k$, in $List_j$ do
                Let $< e_l, l, List_l >$ be the current tag of $R_k$.
                For each rule, $R_l$, in $List_l$ do
                    Append $R_l$ to affectedRuleList.
                    Recursively apply step 3.2.1 for all rules in the tag of $R_l$.
                    Change the tag of $R_l$ to NULL
4. Assign appropriate Gator networks to $R_i$ and the rules in $List_j$.
   Change the Gator network of $R_i$ to $G_{r_j e_j}$ and its tag to $< e_j, j, list_j >$.
   For each rule, $R_k$, in $List_j$ do
       Change the Gator network of $R_k$ to $G_{r_k e_j}$
       Change the tag of $R_k$ to $< e_j, m, list_j - R_k + R_i >$.
```

parameter	value
stopping condition	based on time
local minimum	r-local minimum
next state	random neighbor

Figure 6.14. Parameters for II

5. Find sharing among the rules in *affectedRuleList*.

The local change operator *Swap Gator networks* not only swaps the Gator networks for a selected rule but also tries to find new common shared expressions among the rules that are affected by the swapping. An alternate way to define *Swap Gator networks* could be to swap the Gator networks for a selected rule and then assign optimal Gator networks for the affected rules. We feel that this definition may lead to solutions having a number of rules being assigned with optimal Gator networks, especially when a lot of rules are affected by the swapping operation. Our definition of *Swap Gator networks* will try to find new shared CSEs among the affected rules, resulting in solutions with lower costs. The local change operator defined is complete because by applying it repeatedly any solution can be reached from any other solution in the search space.

6.4.3 Optimizer Tuning

The implementation specific parameters of II that were chosen for the MRO search problem are given in Figure 6.14. These parameters were chosen based on past experience with the algorithms in Gator network optimization (Chapter 4), query optimization [57, 24, 22, 23, 28] and other fields [1, 59]. For deciding the local minimum, the same approximation was used as by Ioannidis [22]. The same approximation was used in Gator network optimization also. A state is considered to be an r-local minimum if the cost of that state is less than that of the cost of n randomly chosen neighbors (with repetition) of that state. Here, n was chosen to be equal to $1.25 * R$ where R is the number of rules in the MRO search problem. We selected this value after extensive experimentation with different values of n. The number of neighbors of a state as examined by this approach can be much less than the actual number of neighbors of a state in this problem. Different states may have different neighbors and deciding a local minimum by exhaustively searching the neighbors of a state is an expensive process and hence we believe that the choice of using an r-local minimum is a more practical one.

In the MRO search problem, a maximum time limit was used as a stopping criterion. A similar criterion was used in query optimization [57]. The maximum time allowed for an experiment was chosen to be proportional to R^2, where R is the number of rules in the experiment. The time limit for 100 rules was set to 60 minutes. The time limit for other experiments is calculated as follows: if x is the number of rules in an experiment then the maximum time allowed for that experiment to run is given by $60(\frac{x}{100})^2$ minutes. Since a time limit is used as the stopping criterion, it is hard to assess the optimality of the solution generated by the algorithm. However, since the same criterion is being used in all the experiments, it allows studying the relative effectiveness of various proposed heuristics.

This concludes the discussion of the proposed approach to solve the multiple rule optimization problem. Experiments have been conducted on randomly generated data to study the effectiveness of the proposed heuristics. Section 6.5 gives the details of the simulator that has been developed to generate the test data for the proposed search algorithm. Section 6.6 gives an analysis of the experimental results.

6.5 MRO Simulator

This section describes the simulator software that has been developed to generate the test data for the proposed randomized search algorithm. The simulator generates rules and common sub-expressions among rules *randomly*, based on the values of the input parameters.

The simulator and the search algorithm were written in C++. Rules and common sub-expressions are represented by their respective identifiers. The output generated by the simulator consists of a list of rule identifiers, a list of CSE identifiers, an RE-list for each rule, an ER-list for each CSE, the costs of optimal Gator networks of rules and the costs of optimal sharable Gator networks of rules corresponding to their CSEs. As explained in section 6.4, an RE-list of a rule is the list of all CSEs that are common between that rule and the other rules in the system and an ER-list of a CSE is the list of all rules that contain that CSE as part of their RCG.

The simulator generates rules and CSEs among rules randomly, based on the values of the input parameters. The costs of optimal Gator networks and the optimal sharable Gator networks of rules are also generated randomly. The details of various input parameters used by the simulator and the values assigned to these parameters are given next.

The simulator takes the following parameters as input:

1. Number of rules
2. Number of contiguous CSEs (Here, CSE refers to a contiguous set of shared edges. Also, it is assumed that different CSEs do not have any common edges. This decision was taken only to simplify the simulator software and this decision will not have any affect on the performance of the search algorithm.)
3. Sharability of a CSE
4. Minimum and Maximum cost of an optimal Gator network
5. Minimum and Maximum cost of a CSE
6. Minimum and Maximum cost of an optimal sharable Gator network

The details about the values assigned to these parameters and the rationale behind choosing those values is given next.

Number of rules. We conducted five sets of experiments. Each set contains $(10 * i)$ rules, where $1 \leq i \leq 5$.

Number of CSEs. Number of CSEs is varied as a function of the number of rules.

$$\text{Number of CSEs} = c * \text{Number of rules (here, } c = 1)$$

Sharability of a CSE. The sharability of a CSE is defined as the percentage of the total number of rules which contain that CSE as a subgraph of their Rule Condition graph.

Sharability is varied from ($\frac{200}{n}\%$ to 100%), where n is the total number of rules in the system. For each CSE, the sharability is selected randomly from the above range. If $s\%$ is the sharability of a CSE then the number of rules which contain that CSE as part of their RCG is given by, $NRules = \lfloor \frac{n*s}{100} \rfloor$. The identifiers of the rules sharing that CSE are decided by randomly selecting $NRules$ from $RSet$, where $RSet$ is the list of all the rules.

$[Min\_opt\_cost,\ Max\_opt\_cost] = [200, 400]$. The terms Min_opt_cost and Max_opt_cost refer to the minimum and maximum cost of an optimal Gator network, respectively. The cost of the optimal Gator network of a rule is selected randomly from the above range. The range was selected to account for the variance in relation sizes, update frequencies of relations and rule sizes. The cost of an optimal Gator network is dependent on all these factors.

$[Min\_cse\_cost,\ Max\_cse\_cost] = [10, 200]$. Min_cse_cost and Max_cse_cost refer to the minimum and maximum cost of a CSE respectively. Here, CSE refers to the contiguous set of edges that are common among a set of rules. The cost of a CSE is decided by choosing randomly from the above range.

[$Min\_opt\_sharable\_Gator\_cost$, $Max\_opt\_sharable\_Gator\_cost$]. The terms $Min\_opt\_sharable\_Gator\_cost$ and $Max\_opt\_sharable\_Gator\_cost$ refer to the minimum and maximum cost of an optimal sharable Gator network of a rule. Let $S = \{s_1, s_2, \dots, s_n\}$ be a set of shared CSEs. Let *cost_S* represent the sum of the costs of all the CSEs in S (the assumption is that CSEs in S do not have any common edges). Let *cost_opt* be the cost of the optimal Gator network for that rule. The optimal sharable Gator network for S will satisfy the following property:

$$Min\_opt\_sharable\_gator\_cost \leq max(cost\_S, cost\_opt)$$

Now, $Max\_opt\_sharable\_gator\_cost$ is selected as follows:

```
if $cost\_S < cost\_opt$ then
    $Max\_opt\_sharable\_gator\_cost = cost\_opt + 0.5 * cost\_S$
else if $(cost\_S > cost\_opt)$ then
    $Max\_opt\_sharable\_gator\_cost = 1.1 * cost\_S$
else
    $Max\_opt\_sharable\_gator\_cost = cost\_opt$
```

The intuition behind choosing these values is the following: sharing a CSE among rules is useful only when the increase in the costs of Gator networks due to the inclusion of shared CSEs (i.e. the difference between the costs of optimal sharable Gator networks and the optimal networks) does not exceed the cost of the shared CSEs.

Hence, when $cost\_S < cost\_opt$, the maximum cost of an optimal sharable Gator network is chosen as $(cost\_opt + 0.5 * cost\_S)$. (To be more precise, if a CSE is shared among N rules then the maximum cost of an optimal sharable Gator network can be chosen as $(cost\_opt + ((N-1)/N) * cost\_S))$.

However, when $cost\_S > cost\_opt$ then the size of the shared CSEs is close to that of the rule size and hence we expect less variance between $cost\_S$ and $Max\_opt\_sharable\_gator\_cost$. Hence, in this case the maximum cost of an optimal sharable is chosen as $(cost\_S + 0.1 * cost\_S = 1.1 * cost\_S)$.

6.6 Experimental Setup

This section presents the details of various experiments conducted to study the effectiveness of the randomized search strategy and the various heuristics presented in Section 6.4. Experiments were conducted on synthetic test data generated by the simulator.

The input to the search algorithm consists of a list of rules represented by their identifiers, a list of contiguous common-subexpressions among these rules, an RE-list for each rule, an ER-list for each CSE, the cost of optimal Gator networks of rules and the costs of optimal sharable Gator networks of rules corresponding to their CSEs.

Experiments were conducted on five different data sets, each generated with a different number of rules. Each data set consisted of i $(i = 10, 20, 30, 40, 50)$ rules. On each data set, 56 experiments were conducted by using different heuristics. In each experiment, the search algorithm was run 5 times with different random seeds. The output of the search algorithm consists of a vector of Gator networks selected for the rules in the system and the CSEs shared among these Gator networks. An *AllOptimal* solution for a set of rules is the solution obtained by choosing an optimal Gator network for each rule. The performance metric in our experiments is the percentage improvement of the solution output by the search algorithm (Iterative Improvement, here) over the *AllOptimal* solution. If X is the cost of the solution generated after multiple-rule optimization and Y is the cost of the *AllOptimal* Solution then the improvement due to sharing of Gator network nodes among the rules is given by:

$$Percentage\ Improvement\ =\ \frac{X-Y}{X}\times 100\%$$

The experiments can be classified into two categories based on heuristics *Select-All* and *Select-A-Few* (details given below). In each of these two categories, the following experiments were conducted: One experiment was conducted by allowing the rules to share only contiguous CSEs and another set (call it Set2) by allowing the rules to share both contiguous CSEs and composite CSEs. Set2 can be classified into three sub-categories based on the way the composite CSEs are generated (Pick-2^k, Pick-k^2 and Pick-k heuristics). In each of these sub-categories, experiments were conducted for all combinations of k values (the value of k is varied from 2 to 5) and the criteria used to generate the composite CSEs (Sort on Cost, Sort on Cost*NoOfRules and Sort on Cost/NoOfRules).

The graphs are plotted as described below. For each data set, there are two graphs based on heuristics *Select-All* and *Select-A-Few*. In each graph, values on the X-axis denote the following: x=1 is for the case with only contiguous CSEs and x=2 to 10 are for the cases with both contiguous CSEs and composite CSEs. x=2 to 5 are for the Pick-k heuristic with k value as 2, 3, 4 and 5 respectively. x=6 to 7 are for the Pick-k^2 heuristic with the k value as 4 and 5 respectively. x=8 and 10 are for the Pick-2^k heuristic with k equal to 3, 4 and 5 respectively. In each experiment, the search algorithm was run 5 times with different random seeds and the average of the percentage improvement of the solution in all the 5 runs was computed. The Y-axis denotes the average percentage improvement of the solution generated by the search strategy for the various experiments that were conducted.

A brief description of the various heuristics that were employed in the proposed search strategy are given next.

Heuristics Select-All and Select-A-Few: The steps followed while generating a random solution include: selecting a rule R_i, selecting a CSE which R_i is going to share with the other rules and selecting the other rules with which R_i is going to share that CSE. This procedure is repeated for all rules in the system (refer to section 6.4 for more details). For each rule, after selecting a CSE and while choosing the other rules to share that CSE, there are two choices: either select all the rules that are eligible to share that CSE (*Select-All*) or select only a subset of those rules that are eligible to share that CSE (*Select-A-Few*).

Two sets of experiments were conducted based on this decision. In one set of experiments, called *Select-All*, all the eligible rules were selected while in the other set, called *Select-A-Few*, only a randomly selected subset of the eligible rules were selected.

No Composite CSEs: Experiments were run by letting the rules share only the contiguous CSEs among them. Each rule can share only a single contiguous CSE with other rules. Rules can not share composite CSEs.

This restriction allows studying the improvement in the solution quality when the composite CSEs are allowed to be shared among the rules.

Pick-2^k: As described in section 6.4, heuristic Pick-2^k allows a maximum of 2^k composite CSEs to be generated for a rule. Experiments are run with different values of k. k is varied from 2 to 4.

Pick-k^2: Pick-k^2 allows a maximum of $k(k-1)/2$ composite CSEs to be generated for each rule, as given in section 6.4. The value of k is assigned to be the same for each rule.

Pick-k: Pick-k allows a maximum of k composite CSEs to be generated for a rule, as given in section 6.4. The value of k was chosen to be the same for all rules.

Greedy Approach: In a random solution generator, for each rule, a CSE is selected randomly from the set of allowed CSEs of that rule. In a greedy solution generator, for each rule, a CSE having the highest cost (from among the set of allowed CSEs of that rule) is selected.

Generating Composite CSEs:

Sort On Cost: The contiguous CSEs of a rule are sorted in non-increasing order based on their cost. Composite CSEs are generated by selecting the first k CCSEs, starting from the highest cost CCSE, in the sorted list and by using the techniques (Pick-2^k, Pick-k^2 and Pick-k) described above.

Sort on Cost*NoOfRules: The contiguous CSEs of a rule are sorted in non-increasing order based on the value of the cost function ($NoOfRules * CSECost$), that was computed for each CCSE of that rule. Composite CSEs are generated by selecting the first k CCSEs, starting from the highest cost CCSE, in the sorted list and by using the techniques (Pick-2^k, Pick-k^2 and Pick-k) described above.

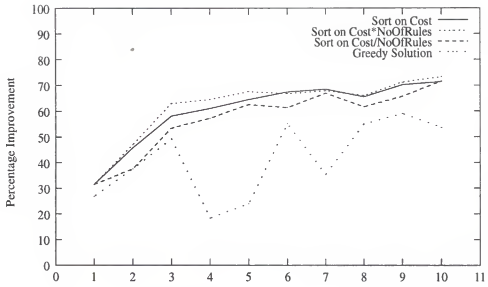
Sort On Cost/NoOfRules: The contiguous CSEs of a rule are sorted in non-increasing order based on the value of the cost function ($CSECost/NoOfRules$) that was computed for each CCSE of that rule. Composite CSEs are generated by selecting the first k CCSEs, starting from the highest cost CCSE, in the sorted list and by using the techniques (Pick-2^k, Pick-k^2 and Pick-k) described above.

6.7 Analysis of Results

Figures 6.15 through 6.19 show the results of various experiments conducted for different rules under different heuristics.

It can be noticed that in all the graphs, the performance of any algorithm that includes composite CSEs (except the greedy solution) is significantly better than the one that shares contiguous CSEs. Hence, it is better for rules to share both contiguous and composite CSEs rather than restricting the choice to only contiguous CSEs.

Among the various criteria that were used to select the basic CSEs, Sort on Cost*NoOfRules performed the best followed by Sort on Cost and Sort on Cost/NoOfRules in that order. The reason for the superiority of Sort on Cost*NoOfRules



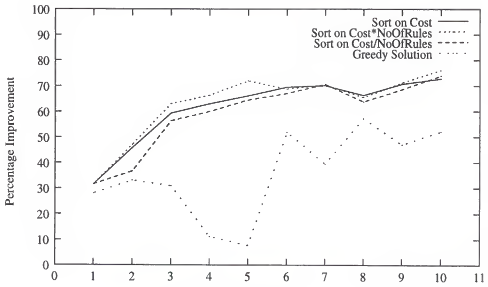
x=1, No composite CSEs

x=2 to 5, Pick-k with k=2,3,4 and 5

x=6 to 7, Pick-k^2 with k=4 and 5

x=8 to 10, Pick-2^k with k=3,4 and 5

(A) Number of Rules: 10, Select-All Heuristic



x=1, No composite CSEs

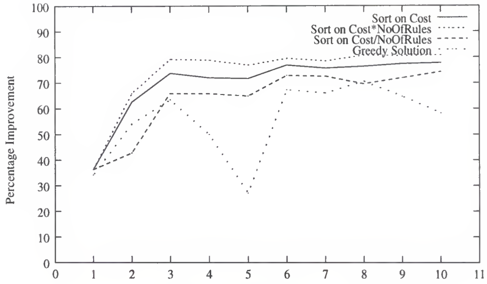
x=2 to 5, Pick-k with k=2,3,4 and 5

x=6 to 7, Pick-k^2 with k=4 and 5

x=8 to 10, Pick-2^k with k=3,4 and 5

(B) Number of Rules: 10, Select-A-Few Heuristic

Figure 6.15. Results of experiments on a rule-set of size 10



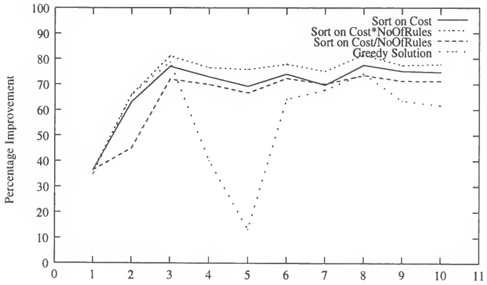
x=1, No composite CSEs

x=2 to 5, Pick-k with k=2,3,4 and 5

x=6 to 7, Pick-k^2 with k=4 and 5

x=8 to 10, Pick-2^k with k=3,4 and 5

(A) Number of Rules: 20, Select-All Heuristic



x=1, No composite CSEs

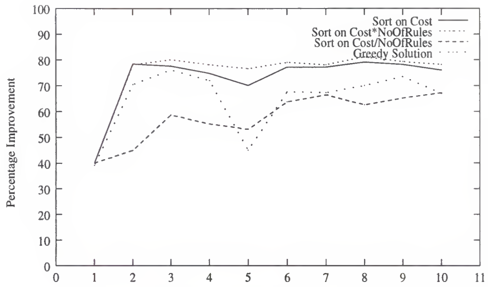
x=2 to 5, Pick-k with k=2,3,4 and 5

x=6 to 7, Pick-k^2 with k=4 and 5

x=8 to 10, Pick-2^k with k=3,4 and 5

(B) Number of Rules: 20, Select-A-Few Heuristic

Figure 6.16. Results of experiments on a rule-set of size 20



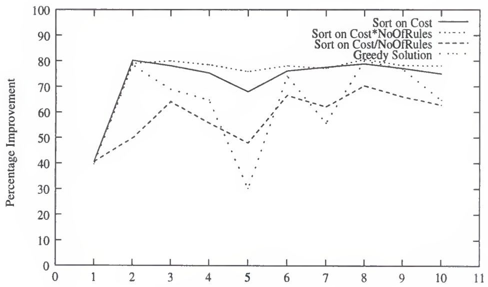
x=1, No composite CSEs

x=2 to 5, Pick-k with k=2,3,4 and 5

x=6 to 7, Pick-k^2 with k=4 and 5

x=8 to 10, Pick-2^k with k=3,4 and 5

(A) Number of Rules: 30, Select-All Heuristic



x=1, No composite CSEs

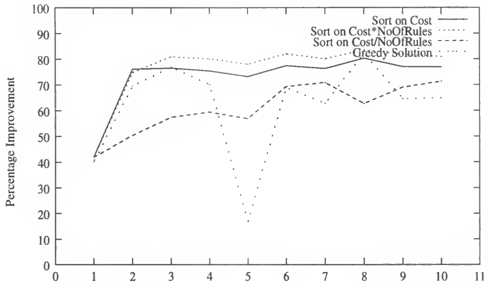
x=2 to 5, Pick-k with k=2,3,4 and 5

x=6 to 7, Pick-k^2 with k=4 and 5

x=8 to 10, Pick-2^k with k=3,4 and 5

(B) Number of Rules: 30, Select-A-Few Heuristic

Figure 6.17. Results of experiments on a rule-set of size 30



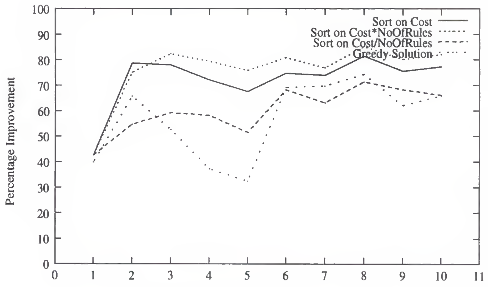
x=1, No composite CSEs

x=2 to 5, Pick-k with k=2,3,4 and 5

x=6 to 7, Pick-k^2 with k=4 and 5

x=8 to 10, Pick-2^k with k=3,4 and 5

(A) Number of Rules: 40, Select-All Heuristic



x=1, No composite CSEs

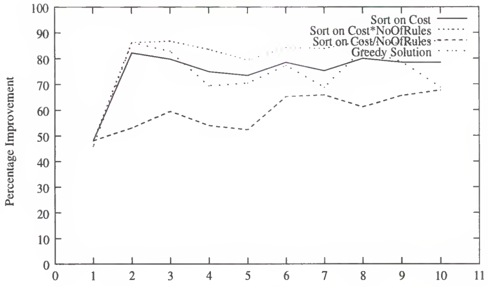
x=2 to 5, Pick-k with k=2,3,4 and 5

x=6 to 7, Pick-k^2 with k=4 and 5

x=8 to 10, Pick-2^k with k=3,4 and 5

(B) Number of Rules: 40, Select-A-Few Heuristic

Figure 6.18. Results of experiments on a rule-set of size 40



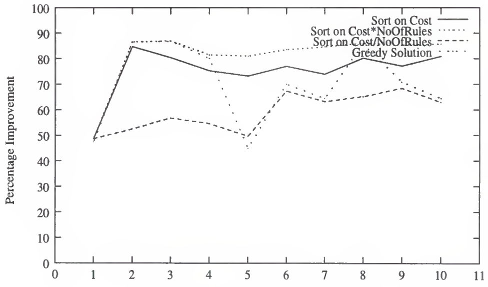
x=1, No composite CSEs

x=2 to 5, Pick-k with k=2,3,4 and 5

x=6 to 7, Pick-k^2 with k=4 and 5

x=8 to 10, Pick-2^k with k=3,4 and 5

(A) Number of Rules: 50, Select-All Heuristic



x=1, No composite CSEs

x=2 to 5, Pick-k with k=2,3,4 and 5

x=6 to 7, Pick-k^2 with k=4 and 5

x=8 to 10, Pick-2^k with k=3,4 and 5

(B) Number of Rules: 50, Select-A-Few Heuristic

Figure 6.19. Results of experiments on a rule-set of size 50

criterion appears to be that the frequently used CSEs with higher cost may have been shared by all or most of the rules in their RE-lists in good solutions. Remember that the RE-list of a CSE is the list of rules that have that CSE as a part of their rule condition graphs.

Another observation is that there is no significant difference in the performance of different algorithms under select-All and select-A-Few heuristics. This heuristic does not seem to be a significant factor in recognizing the sharing among rules. The reason for this could be that CSEs may have been shared by most of the rules in good solutions even under Select-A-Few heuristic. As mentioned in the previous paragraph, this could also the reason why the Sort on Cost*NoOfRules criterion performed better than the others.

There is a wide variation in the quality of the solutions generated by the greedy algorithm. The percentage improvement of the greedy solution is in the range [7.7% to 86.7%] (in all the experiments). Its performance is worse than that of the one generated by Sort on Cost/NoOfRules in most of the cases and is close to that of Sort on Cost*NoOfRules or Sort on Cost in a few cases. We noticed a wide variation in the solution cost even among the different runs (run with different seeds) of the same experiment. Another observation regarding the greedy solution is that it generated a very bad solution consistently for the Pick-k criterion with $k = 5$. The reason for its behavior for this case is not clear.

Figure 6.15 shows the results of experiments conducted on 10 rules. Here, it can be noticed that the performance of Pick-2^k is the best followed by Pick-k^2 and then Pick-k. Also, within each category, the solution quality increased with k. The superiority of Pick-2^k is expected because Pick-2^k allows all possible combinations of the selected contiguous CSEs to be shared among the rules. Pick-k^2 allows more composite CSEs than Pick-k but less than the ones allowed by Pick-2^k and hence its performance is in between that of Pick-k and Pick-2^k. Within each category, an increase in k allows more composite CSEs to be shared among the rules.

In Figure 6.16, among the experiments conducted on 20 rules, the relative performance among Pick-k, Pick-k^2 and Pick-2^k is similar to the one in experiments on 10 rules. However, within each category, the solution quality remained almost the same for different values of k.

In Figures 6.17, 6.18 and 6.19, among the experiments conducted on 30, 40 and 50 rules respectively, there is no significant different in the performance of Pick-k, Pick-k^2 and Pick-2^k. Also, within each category, the solution quality either remained the same or decreased with k.

The reason for the relative behavior of Pick-k, Pick-k^2 and Pick-2^k among 30, 40 and 50 rules seems to be because of the increase in the search space (the search space increases exponentially with the number of rules) in these experiments. The difference in the CSEs generated by Pick-k, Pick-k^2 and Pick-2^k is not significant enough to make a big difference in the solution quality.

It can be noticed even though there is a significant difference in the number of composite CSEs generated between Pick-k and Pick-2^k, the difference in the solution quality generated by these two algorithms is not that significant (the difference in the quality of solution generated by Pick-k and Pick-2^k is 7% in case of 10 and 20 rules and almost the same in the other cases).

In the proposed architecture for MRO, an optimal sharable Gator network needs to be generated for each rule for each of its composite CSEs and since generating a optimal sharable Gator network takes considerable amount of time, Pick-k is the option that is best over the widest number of rules. When the number of rules is low, it is advisable to use Pick-2^k heuristic.

Since the value of the optimal solution is not known in these experiments, it is hard to comment on the optimality of the solutions generated by the randomized search algorithm in precise terms. But, since the Percentage Improvement in most of the experiments run using the heuristic Sort On Cost*NoOfRules is in [60%,87%] range, it can be said that it is generating a reasonably good solution.

6.8 Conclusion

Multiple Rule Optimization (MRO) is the problem of finding Gator networks for a set of rules such that the total cost of pattern matching (the total cost of Gator networks for all rules with the cost of shared nodes counted only once) for all the rules is minimal. This chapter proposed a new approach to solve the multiple rule optimization problem.

The proposed approach makes use of the existing Gator network optimizer to generate a locally optimal Gator network for each rule. In addition, it generates a set of sub-optimal Gator networks, called locally optimal sharable Gator networks, for each rule. The number of locally optimal sharable Gator networks generated for a rule depends on the number of shared expressions between that rule and the other rules in the system. Finally, search is done among the search space consisting of locally optimal and the locally optimal sharable Gator networks of rules to find a globally optimal solution.

Next, a search strategy based on randomized algorithms has been proposed. Heuristics have been suggested to reduce the search space. Experiments have been conducted on a randomly generated test data to study the effectiveness of the search strategy and the various proposed heuristics.

Experimental results show that the search strategy based on randomized algorithms generates reasonably good solutions (the Percentage Improvement of most of the solutions generated by using the heuristic 'Sort On Cost*NoOfRules' was in the range [60%,87%]). 'Sort On Cost*NoOfRules' and 'Pick-K' are the suggested heuristics for a system consisting of a large number of rules.

CHAPTER 7
SUMMARY AND FUTURE RESEARCH DIRECTIONS

7.1 Summary

This dissertation explored the discrimination network-based approach for pattern matching of multi-join triggers in active databases. The first part of this dissertation (Chapters 2, 3, 4 and 5) investigated the problem of finding an optimal generalized discrimination network for a trigger. The second part of this dissertation (Chapter 6) investigated the problem of finding a globally optimal set of discrimination networks for a collection of triggers. This chapter reiterates the results of this dissertation and discusses some avenues for future research.

A major conclusion of this work is that optimized Generalized discrimination networks (or optimized Gator networks) normally have a shape which is neither pure Rete nor TREAT, but an intermediate form having a few beta nodes (less than the number of beta nodes in a Rete network). The fan-out of beta nodes in optimized Gator networks depend on the amount of buffer space that is available. In a plentiful-buffer environment, optimized Gator networks are observed to have a fan-out of two to four. In a low-buffer environment, the fan-out is higher. This means that, optimized Gator networks have fewer beta nodes in a low-buffer environment than in a plentiful-buffer environment.

The performance of Gator networks is significantly better than that of TREAT and reasonably better than that of Rete in a plentiful-buffer environment. In a TREAT network, pattern matching involves performing join operations with all the alpha nodes in the network. Gator and Rete network take advantage of the pre-computed results in the beta nodes to avoid some of these join operations. Also, the maintenance cost of beta nodes in Gator and Rete is low because of the availability of large buffer space (cost model 1). These factors contribute to the superiority of Gator and Rete over TREAT in a plentiful-buffer environment. In case of Gator, the optimizer has the flexibility in choosing the number of beta nodes depending on the update frequency and other database statistics. Only those beta nodes that are useful are created in an optimized Gator network. This explains the superiority of Gator over a Rete network. These results also show that, even in a plentiful buffer environment, it is beneficial to have fewer beta nodes than the ones in a Rete network.

Optimizer results indicate that, Gator networks perform better than Rete and TREAT in a low-buffer environment also. In this environment, it was observed that, TREAT networks have lower cost than Rete networks. Here, the cost of maintaining beta nodes is significant and that explains the higher fan-out of beta nodes (which means fewer beta nodes in the network) and the superiority of TREAT over Rete. Here, even though TREAT was favored over Rete, optimized Gator networks had a few beta nodes and they were not pure TREAT networks. Overall, it is beneficial to use optimized Gator networks rather than restricting the possibilities to traditional Rete and TREAT networks.

In general, the amount of available buffer space and the update frequency distribution have a noticeable effect on the shape of optimized Gator networks. For equal frequency distribution, the shape of the optimized Gator network is balanced with almost equal root-to-leaf path length for all leaf nodes. For step and skew frequencies, the shape of the optimized Gator network is skewed, with the relations having higher update frequencies placed closer to the P-node. It was also noticed that the update frequency does not totally dominate other factors such as data size, condition graph shape and predicate selectivity. Since bushy trees allow more flexibility in the shape of a binary tree network (than left-deep binary trees), the performance of bushy trees is expected to closely match that of Gator in this environment.

Validation of Gator network cost functions have shown that join selectivity is a crucial factor in estimating the cost of a Gator network and that it has a significant effect on the accuracy of the estimated cost of a Gator network.

This work not only demonstrates the superiority of Gator networks over Rete and TREAT but also shows the feasibility of finding an optimized Gator network in a reasonable amount of time by using randomized search algorithms. This is another significant contribution of this work. Three randomized algorithms II, SA and TPO were explored and it was demonstrated that TPO performs better, in terms of output quality, compared to the other two algorithms for the Gator network optimization problem. The output quality of the algorithms was close, but TPO was best or second best in every test. TPO required a lot of tuning effort compared to the other two algorithms and its performance was very sensitive to the initial temperature of the

SA phase, in addition to the number of local optimizations of the II phase. Because the optimization time required for all three algorithms was relatively close, and II is fairly simple to tune, implementors of Gator network optimizers may want to choose II to reduce the effort required to tune the optimizer.

This work has clearly demonstrated the superiority of Gator over Rete and TREAT networks. In conclusion, optimized Gator networks can help make it possible to improve the trigger condition processing capability of future database systems.

Another contribution of this dissertation is in proposing a new strategy to solve the Multiple Rule Optimization problem. A search strategy based on randomized algorithms, along with a set of heuristics to reduce the search space, has been presented to find an efficient global strategy for a set of rules. Experimental results show that the search strategy based on randomized algorithms generates reasonably good solutions (the percentage improvement of most of the solutions generated by using the heuristic 'Sort On Cost*NoOfRule' was in the range [60%,87%]). 'Sort On Cost*NoOfRules' and 'Pick-K' are the suggested heuristics for a system consisting of a large number of rules. More experiments need to be conducted on real world data to assess the amount of shared computation in trigger conditions, as it determines the benefits achieved through MRO.

7.2 View Maintenance Using Discrimination Networks

Discrimination networks can also be used to maintain materialized views in a DBMS. When a discrimination network is used for view maintenance, the P-node represents the materialized view for the view condition and the alpha and beta nodes represent the extra views that need to be materialized in order to efficiently maintain the given view. Hence, optimized Gator networks provide an efficient way to maintain materialized views also. The work of Ross et al. [43] also addresses the problem of what additional views to materialize in order to efficiently maintain a materialized view. However, they have presented only an exhaustive search algorithm for this problem.

Similarly, the work on multiple rule optimization can be used to efficiently maintain a collection of materialized views. Thus, the research work presented in this dissertation can also be used to efficiently maintain views and implement assertions in a DBMS.

7.3 Future Research Directions

One potential avenue for future research is extending the Gator networks to handle aggregation operators. Right now, Gator networks handle rule conditions having select and join operators only. This improvement involves extending the cost functions and the optimization strategy of Gator networks to handle aggregation operators. The work of Ross et al. [43] is a starting point in this direction. Their work, which was developed in the context of view materialization, allows aggregation operators but provides only an exhaustive optimization strategy and it is not likely to scale to rules with large number of relations.

Another important problem that needs to be looked at is deciding when and how to re-optimize a Gator network for a rule. The database statistics that are used in building a Gator network may become stale over time and a new Gator network may need to be found. This is an important problem because if the statistics change, an optimized Gator network (optimized with the old statistics) may not be efficient any more and that might result in poor performance.

Another interesting problem in the context of improving the performance of a Gator network is extending the status of alpha and beta nodes to include a caching strategy. Right now, the tuples of a stored alpha node or a beta node are assumed to be available on disk. The tuples of frequently touched nodes (either alpha or beta) could be cached in main memory to improve the performance. The update frequency distribution of relations is a valuable input here because it can be used to infer what nodes in a Gator network are frequently touched during token propagation.

Finally, in the context of MRO, it would be interesting to compare the performance and optimization time of the proposed randomized search strategy with that of Shim & Sellis's A*-algorithm [46] and the dynamic programming approach of Park and Segev [26].

REFERENCES

[1] Emile Aarts and Jan Korst. *Simulated Annealing and Boltzmann Machines.* John Wiley and Sons, 1990.

[2] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L.Reeve, and Jr. James B.Rothnie. Query processing in a system for distributed databases (sdd-1). *ACM Transactions on Database Systems*, 6(4):602–625, December 1981.

[3] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: an Introduction to Rule-Based Programming.* Addison Wesley, 1985.

[4] M. J. Carey, D. J. DeWitt, and Scott L. Vandenberg. A data model and query language for EXODUS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 413–423, June 1988.

[5] Stefano Ceri, Pietro Fraternali, Stefano Paraboschi, and Letizia Branca. Active rule managment in Chimera. In Stefano Ceri and Jennifer Widom, editors, *Active Database Systems: Triggers and Rules for Advanced Database Processing*, pages 151–176. Morgan Kaufmann, 1996.

[6] Stefano Ceri and Giuseppe Pelagatti. *Distributed Databases.* McGraw-Hill computer science series, 1984.

[7] Ullas Chadaga. Performance evaluation and tuning of optimized active database discrimination networks. Master's thesis, University of Florida, CIS Department, May 1998.

[8] S. Chakravarthy, B. Blaustein, A. P. Buchmann, M. Carey, U. Dayal, D. Goldhirsch, M. Hsu, R. Jauhari, R. Ladin, M. Livny, D. McCarthy, R. McKee, and A. Rosenthal. HiPAC: A research project in active, time-constrained database management, Final Technical Report. Technical Report XAIT-89-02, Xerox Advanced Information Technology, August 1989.

[9] Sharma Chakravarthy. Divide and conquer: A basis for augmenting a caonventional query optimizer with multiple query processing capabilities. In *Proceedings of the 7th International Conference on Data Engineering Conference*, pages 482–490, April 1991.

[10] S. Finkelstein. Common expression analysis in database applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 235–245, 1982.

[11] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[12] N. Gehani and H.V. Jagadish. Ode as an active database: Constraints and triggers. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, September 1991.

[13] Eric N. Hanson. Gator: A generalized discrimination network for production rule matching. In *Proceedings of the IJCAI Workshop on Production Systems and Their Innovative Applications*, August 1993.

[14] Eric N. Hanson. The design and implementation of the Ariel active database rule system. *IEEE Transactions on Knowledge and Data Engineering*, 8(1):157–172, February 1996.

[15] Eric N. Hanson, Sreenath Bodagala, Mohammed Hasan, Goutam Kulkarni, and Jayashree Rangarajan. Optimized rule condition testing in ariel using gator networks. Technical Report TR-95-027, University of Florida CIS Dept., October 1995. http://www.cis.ufl.edu/cis/tech-reports/.

[16] Eric N. Hanson, Moez Chaabouni, Chang-ho Kim, and Yu-wang Wang. A predicate matching algorithm for database rule systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 271–280, May 1990.

[17] Eric N. Hanson and Mohammed S. Hasan. Gator: An optimized discrimination network for active database rule condition testing. Technical Report TR-93-936, University of Florida CIS Dept., December 1993. http://www.cis.ufl.edu/cis/tech-reports/.

[18] Mohammed Hasan. Optimization of discrimination networks for active databases. Master's thesis, University of Florida, CIS Department, November 1993.

[19] Linda Howe. Sybase data integrity for on-line applications. Technical report, Sybase, Inc., Emeryville, CA, 1986.

[20] INGRES Corporation. *INGRES/SQL Reference Manual*, November 1989. Version 6.3.

[21] Yannis E. Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 268–277, 1991.

[22] Yiannis Ioannidis and Younkyung Cha Kang. Randomized algorithms for optimizing large join queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 312–321, May 1990.

[23] Yiannis Ioannidis and Younkyung Cha Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 168–177, May 1991.

[24] Yiannis Ioannidis and Eugene Wong. Query optimization by simulated annealing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1987.

[25] Toru Ishida. An optimization algorithm for production systems. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):549–558, August 1994.

[26] J.Park and A.Segev. Using common subexpressions to optimize multiple queries. In *Proceedings of IEEE Data Engineering Conference*, pages 311–319, February 1988.

[27] Mokhtar Kandil. *Support for Expensive Selection Predicates in Active Database Discrimination Networks.* PhD thesis, University of Florida, Gainesville, (In preparation).

[28] Younkyung Cha Kang. *Randomized Algorithms for Query Optimization.* PhD thesis, University of Wisconsin, 1991.

[29] S. Kirkpatrick, C. C. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[30] Donald E. Knuth. *Fundamental Algorithms.* Addison Wesley, 1973.

[31] Goutam Kulkarni. Extending the Ariel active DBMS with Gator, an optimized discrimination network for rule condition testing. Technical Report TR95-006, University of Florida, CIS Dept., February 1995. MS thesis, http://www.cis.ufl.edu/cis/tech-reports/.

[32] J. McDermott, A. Newell, and J. Moore. The efficiency of certain production system implementations. In *Pattern-directed Inference Systems.* Academic Press, 1978.

[33] M.Garey and D.S.Johnson, editors. *Computers and Intractability - A Guide to Theory of NP-Completeness.* W.H.Freeman and Company, 1979.

[34] Daniel P. Miranker. TREAT: A better match algorithm for AI production systems. In *Proc. AAAI National Conference on Artificial Intelligence*, pages 42–47, August 1987.

[35] M.Stonebraker, A.Jhingran, J.Goh, and S.Potamianos. On rules, procedures, caching and views in data base systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 281–290, Atlantic City, NJ, May 1990.

[36] S. Nahar, S. Sahni, and E. Shragowitz. Simulated annealing and combinatorial optimization. In *Proc. of the 23rd Design Automation Conference*, pages 293–299, 1986.

[37] Koyoshi Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proceedings of the 1990 VLDB Conference*, 1990.

[38] Oracle Corporation. *Oracle 7.1 Reference Manual*, 1994.

[39] Jayashree Rangarajan. A randomized optimizer for rule condition testing in active databases. Master's thesis, University of Florida, CIS Department, December 1993.

[40] Joel E. Richardson and Michael J. Carey. Programming constructs for database system implementation in EXODUS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 208–219, May 1987.

[41] Joel E. Richardson, Michael J. Carey, and Daniel T. Schuh. The design of the E programming language. *ACM Transactions on Programming Languages and Systems*, 15(3), 1993.

[42] A. Rosenthal and U.S. Chakravarthy. Anatomy of a modular multiple query optimizer. In *Proc. of VLDB Conf.*, pages 230–239, 1988.

[43] Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 447–458, 1996.

[44] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, June 1979. (reprinted in [51]).

[45] Timos Sellis. Global query optimization. *ACM TODS*, 13(1):23–52, 1988.

[46] Kyuseok Shim, Timos Sellis, and Dana Nau. Improvements on a heuristic algorithm for multiple-query optimization. *IEEE Transactions on Knowledge and Data Engineering*, pages 197–222, 1994.

[47] M. Stonebraker, E. N. Hanson, and C. Hong. The design of the POSTGRES rule system. In *Proceedings of the 1987 IEEE Data Engineering Conference*, pages 365–374, 1987.

[48] M. Stonebraker, M. Hearst, and S. Potaminos. A commentary on the POSTGRES rules system. *SIGMOD Record*, 18(3):5–11, September 1989.

[49] M. Stonebraker and G. Kemnitz. The POSTGRES next-generation database management system. *Communications of the ACM*, 34(10):78–92, October 1991.

[50] M. Stonebraker and L. Rowe. The design of POSTGRES. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1986.

[51] Michael Stonebraker, editor. *Readings in Database Systems.* Morgan Kaufmann, 1994.

[52] Michael Stonebraker, Eric Hanson, and Spiros Potamianos. The POSTGRES rule manager. *IEEE Transactions on Software Engineering*, 14(7):897–907, July 1988.

[53] Michael Stonebraker, Lawrence Rowe, and Michael Hirohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(7):125–142, March 1990.

[54] S. Y. W. Su and H. Lam. An object-oriented knowledge base management system for supporting advanced applications. In *Proc. of the 4th Int'l Hong Kong Computer Society Database Workshop*, pages 3–22, Dec. 1992.

[55] S. Y. W. Su, H. Lam, S. R. Eddula, J. Arroyo, N. Prasad, and R. Zhuang. OSAM*.KBMS: An object-oriented knowledge base management system for supporting advanced applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 540–541, May 25-28 1993.

[56] Stanley Y.W. Su, Herman Lam, Javier Arroyo-Figueroa, Tsae-Feng Yu, and Zhidong Yang. An extensible knowledge base management system for supporting rule-based interoperability among heterogeneous systems. In *Proc. of Conference on Information and Knowledge Management*, pages 1–10, Nov. 28-Dec. 2 1995.

[57] A. Swami and A. Gupta. Optimization of large join queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 8–17, 1988.

[58] S.Y.W.Su, H.Lam, T.F.Yu, J.Arroyo, Z.Yang, and S.Lee. Ncl: A common language for achieving rule-based interoperability among heterogeneous systems. *Journal of Inelligent Information Systems, Special Issue*, 1995.

[59] P.J.M. van laarhoven and E.H.Aarts. *Simulated Annealing: Theory and Applications.* D.Reidel Publishing Company, 1987.

[60] Yu-wang Wang and Eric N. Hanson. A performance comparison of the Rete and TREAT algorithms for testing database rule conditions. In *Proc. IEEE Data Eng. Conf.*, pages 88–97, February 1992.

[61] Jennifer Widom. The starburst rule system. In Stefano Ceri and Jennifer Widom, editors, *Active Database Systems: Triggers and Rules for Advanced Database Processing*, pages 87–109. Morgan Kaufmann, 1996.

[62] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann, 1996.

BIOGRAPHICAL SKETCH

Sreenath Bodagala was born on January 16, 1972, in Proddatur, Cuddapah (District), Andhra Pradesh, India. He received his Bachelor of Engineering degree in computer science and engineering from Osmania University, Hyderabad, India in June, 1992. He received his Master of Technology degree in computer science and engineering from the Indian Institute of Technology, Bombay, India in January, 1994. He will receive his Doctor of Philosophy degree in computer and information science and engineering from the University of Florida, in August 1998. He is a member of Tau Beta Pi (National Engineering Honor Society, USA). His research interests include active database systems, object-oriented database systems, parallel database systems, and decision-support systems.